
kubuntu
LibreOffice

F-35 - Zasady Kodowania - komentarz

autor:	Jacek Marcin Jaworski
pseudonim:	Energo Koder Atlant
utworzono:	**2022-11-01**
wersja: 2341 z dnia:	**2025-02-09**
Miejsce:	**Pruszcz Gd., Polska**
system:	**Linux (distro: Kubuntu)**
program:	**LibreOffice**

Spis treści

Wstęp

Celem napisania tego dok. jest skomentowanie dok. pt. „Joint Strike Fighter Air Vehicle C++ Coding Standards For The System Development And Demonstration Program” opracowany w firmie Lockheed Martin w nieznanym miejscu na terenie SZAP, jest on datowany na gru. 2005. Ten dok. od lat krąży po sieci Internet jako „F-35 - Coding Rules.pdf”. W tym dok. omówione są zasady kodowania programów w C++ w projekcie samolotu bojowego F-35

zaprojektowanego i aktualnie produkowanego w SZAP (i w niektórych państwach sojuszniczych).

Należy zaznaczyć, że ten dok. „F-35 - Coding Rules.pdf” zawiera wył. wymagania i wskazówki dotyczące samego kodowania. Nie ma w nim prawie nic jak należy prowadzić cały proj. prog. Częściowo ten temat podjąłem w [arch-prog-nieup].

Postanowiłem zmierzyć się z „F-35 - Coding Rules.pdf” z nast. powodów:

1. Po ponad 25l. prog. w j. C++ czuję się na siłach by podjąć takie wyzwanie;
2. Potencjalnie wojskowe, inżynierskie metody wytwarzania są najbardziej zaawansowane;
3. Potencjalnie przemysł wojskowy jest najnowocześniejszy;
4. Potencjalnie SZAP jest liderem w technice i w programowaniu;
5. „F-35 - Coding Rules.pdf” to jedyny techniczny dokument z kręgów wojskowych jaki obecnie (w sty. 2025) jest mi dostępny;
6. III Rzecz (pospolita) nabyła samoloty bojowe F-35, cytat z [f-35-arch-s-w-wiki]; „Polska Siły Powietrzne – 31 stycznia 2020 Polska zamówiła 32 myśliwce F-35, bez wcześniejszego przetargu ani konkursu[230]. Wartość zamówienia wraz z osprzętem naziemnym, pakietem logistycznym i szkoleniowym wynosi 4,6 miliarda dolarów (ok. 17,8 mld zł)[230]. Pierwsze myśliwce mają trafić do Polski w 2024 roku[230]. Mają wejść na wyposażenie 32 BLT w Łasku oraz 21 BLT w Świdwinie[70].”. Bez przetargu ani bez konkursu, czyli jak zwykle „polski wałek”. Co ciekawe s. WWW do której się odnosi ww. s. w wiki, czyli źródło [230], czyli [umo-na-f-35-podpisana-def24], w d. 2025-01-18, nie., już nie ma tej informacji.
7. Moje wcześniejsze notatki z lektury dok. „F-35 - Coding Rules.pdf” były mało wartościowe.

Każdą zasadę opatrzyłem komentarzem.

Każdą zasadę oceniłem wg skali: rewelacja, zaleta, wada, partactwo.

Każda zasadę zaszeregowałem pod względem zastosowania:

1. Do zapamiętania;
2. Przegląd kodu: Oznacza, że należy ją mieć na liście rzeczy do spr. w trakcie przeglądu kodu;
3. Skrypt: kod skanowany lokalnie, statycznie, skryptem przed zatwierdzeniem zmian w repo;
4. Testy automatyczne: dynamiczne testy jednostkowe uruchamiane lokalne lub/i na serwerze ciągłej integracji;
5. BRAK: Oznacza, że zasadą nie należy się przejmować.

Można dodać, że do momentu rozp. proj. F-35 j. C++ był całkowicie cywilnym j. prog. Wcześniej do celów wojskowych w SZAP stosowano j. ADA (tak było nawet w przypadku proj. samolotu F-22, który był zaprojektowany tuż przed rozp. prac nad F-35). Obecnie wiadomo, że jest jeszcze gorzej bo kod C++ trafił już do rakiet jakimi się morduje ludzi: z [ntw-2024-11] wiadomo, że cytat s. 45: „I tak zmiany wprowadzone w pierwszym z wariantów pocisku JASSM-ER [prawdopodobnie chodzi o AGM158 B2 – przyp. JMJ] dotyczyć mają przede wszystkim wymiany komponentów teraz trudnodostępnych, wychodzących z produkcji czy też po prostu przestarzałych (według obecnych standardów). Systemy komputerowe pocisku otrzymały ponadto oprogramowanie, stworzone w języku C++, które zastąpiło wcześniej stosowane, napisane w języku ADA. Pocisk w wariancie AGM-158B-2 wyposażono również w zmodernizowany komputer pokładowy (Missile Control Unit, MCU), dysponujący zwiększoną mocą obliczeniową.”.

Można też dodać, że od momentu upublicznienia „F-35 - Coding Rules.pdf” nastąpiło nagłe przyspieszenie w rozwoju standardu j. C++ i jego bibl. standardowej. Wcześniej opracowano jedynie standard C++98.

Można też dodać, że wkrótce po upublicznieniu „F-35 - Coding Rules.pdf” (w 2008r.) sprzedano norweską firmę Trolltech fińskiemu korpo Nokia i od tamtej pory praktycznie nie rozwija się już bibl. Qt, a jest ona najważniejszą bibl. dla cywilnych programistów C++.

Można też dodać, że mniej więcej od momentu upublicznienia „F-35 - Coding Rules.pdf” ktoś dąży by wyeliminować j. C++ z użycia w cywilu. Programy na PC zastępuje się s. WWW. Natomiast apki na sprytne tel. są pisane wył. w j. skryptowych.

Można dodać, że ja jako cywilny programista C++ od 2022-05-27, pią. do d. dzisiejszego, czyli 2025-02-09, nie. nie mogę znaleźć pracy mimo wysłania 186 podań o pracę do firm na terenie całej Polski. To też pokazuje jak zła jest sytuacja j. C++ na rynku cywilnym. Zaznaczam, że nie wysyłam podań do firm zbrojeniowych, bo ja chcę budować cywilizację a nie spuszczać ludziom bomby na głowę. Przebranżowienie z kolei nie wchodzi w grę, bo na PG (czyli na mojej polibudzie) nie ma możliwości studiowania zaocznie ani wieczorowo żadnego

nowoczesnego kier. poza informą – nie można studiować zaocznie ani wieczorowo ani elektroniki ani telekomunikacji, ani automatyki ani robotyki, ani mechatroniki ani nawet fizyki technicznej (mimo, że na studiach dziennych na PG te kierunki są dostępne). Jest to sprzeczne z [powszechna-deklaracja-praw-człowieka], cytat art. 26, p.1: „[...] Wykształcenie techniczne i zawodowe ma być powszechnie dostępne, a wykształcenie wyższe ma być dostępne na równi dla wszystkich na podstawie rzeczywistych osiągnięć.”. I mimo że mam legalną, nową maturę, to obecnie (w luym 2025r.) nic nie mogę zrobić by studiować prawdziwą technikę na PG, by zmienić zawód i by dalej uczciwie zarabiać na życie.

3.1 Skróty

abs.	abstrakcyjny
alg.	algorytm
art.	artykuł
bit.	bitów
def.	definicja
dł.	długość
dom.	domyślny
el.	element
ew.	ewentualnie
f.	funkcja
gen.	generowany
HJ	Hitlerjugend
itp.	i tym podobne
j. ang.	język angielski
jw.	Jak wyżej
kat.	katalog
kl.	klasa
kol.	kolejny
l.	liczba
l. cał.	liczby całkowite
l. rze.	liczby rzeczywiste
o.	obiekt
org.	oryginalny
pam.	pamięć
poj.	pojedynczy
poł.	położenie
pow.	powyżej
pt.	pod tytułem
roz.	rozdział
s.	strona
spr.	sprawdzenie
SQL	język skryptowy używany w relacyjnych bazach danych
sys. op.	system operacyjny
SZAP	Stany Zjednoczone Ameryki Północnej
tab.	tablica
tel.	telefon
utw.	utworzono
wej.	wejście
wsk.	wskaźnik
wyd.	wydawca, wydanie
wyj.	wyjście
wył.	wyłącznie
zaw.	zawartość
zm.	zmienna

4 Projekt Generalny

Cele jakościowe tworzonego kodu[1]:

1. Program powinien działać spójnie w przewidywany sposób;
2. Kod źródłowy powinien być przenośny (przynajmniej w zakresie kompilatorów i linkerów);
3. Kod powinien być spójny, czytelny, mieć prostą architekturę i być łatwy do uruchamiania ze śledzeniem[2];
4. Kod powinien ułatwiać testowanie. Ułatwia to:
   4.1. Minimalizacja rozmiaru kodu;
   4.2. Minimalizacja złożoności;
   4.3. Minimalizacja l. rozgałęzień w kodzie.
5. Kod powinien być możliwy do ponownego użycia;
6. Wymagania mogą się zmieniać w trakcie życia projektu;
7. [Ponowienie postulatu czytelnego kodu. - przyp. JMJ]

4.1 Rozmiar Kodu i Złożoność

Wymaganie Zwykłe 1.: Zabrania się f. dłuższych niż 200 linii.

1 W j. ang. the code produced.
2 W j. ang. debuger.

Komentarz 1.: To dobra zasada - ale może być lepsza: F. o dł. 200 linii i tak są za długie. F. nie powinna być dłuższa niż ekran kodu, czyli 50-100 linii.

Uzasadnienie: W ten sposób jednym spojrzeniem można ogarnąć całą f.

Wyjątek: f. do parsowania opcji, serializacji danych w bazach i protokołach sieciowych.

Ocena: Zaleta.

Zastosowanie: Przegląd kodu.

Wymaganie Specjalne 2.: Zabrania się programowania samomodyfikującego kodu.

Komentarz 2.: To dobra zasada - ale wydumana: nigdy nie spotkałem się z "samomodyfikującym kodem" w C++. Czytałem jedynie o samomodyfikujących się wirusach.

Ja mam zasadę, że nawet własne generatory kodu zawsze uruchamiam pod nadzorem, bo rzeczywistość zawsze zaskakuje.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 3.: Zabrania się więcej niż 20 warunków log. w f. Wyjątkiem są f. z dopasuj[3] z dużą l. wzorów[4].

Komentarz 3.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

5 Standardy Kodowania w C++

5.1 Wstęp

[Powielenie 4]

5.2 Zasady

5.2.1 Zalecenia, Wymagania Zwykłe, Wymagania Krytyczne

1. Zalecenia (w org. W j. ang. should): dobre rady, jakie należy uwzględniać w miarę możliwości;
2. Wymagania Zwykłe (w org. W j. ang. will): obowiązkowe i nie wymagające weryfikacji;
3. Wymagania Specjalne (w org. W j. ang. shall): obowiązkowe i wymagające weryfikacji.

5.2.2 Zasady Łamania Zasad

Wymaganie Specjalne 4.: Złamanie Zasad musi być zatwierdzone przez:

1. Lidera zespołu inżynierów.

Wymaganie Specjalne 5.: Złamanie Wymagania Zwykłego lub Krytycznego musi być zatwierdzone przez:

1. Lidera zespołu inżynierów;
2. Kierownika produktu.

Komentarz 4. i 5.: To dobra zasada. Jednak nie spotkałem się z taką zasadą w żadnym projekcie w jakim brałem udział.

Ocena: Rewelacja.

Zastosowania: Przegląd kodu.

Wymaganie Specjalne 6.: Odstępstwa od Wymagań Specjalnych należy dokumentować w miejscach występowania (w plikach). Od tej zasady nie ma odstępstw.

Komentarz 6.: To dobra zasada. Jednak nie spotkałem się z taką zasadą w żadnym projekcie w jakim brałem udział.

Ocena: Rewelacja.

Zastosowanie: Przegląd kodu.

5.2.3 Wyjątki Od Stosowania Zasad

Niektóre reguły mają w sobie zawarte klauzule określające kiedy nie trzeba się do nich stosować. W przypadkach określonych tymi klauzulami nie trzeba uzyskiwać zgody na odstępstwo od reguły.

Wymaganie Zwykłe 7.: Dopuszcza się odstępstwo od Zasady lub Wymagania Zwykłego bez zatwierdzania przez przełożonych gdy jest ono określone klauzulą opisującą wyjątek.

Komentarz 7.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.3 Słownik Pojęć

[Lista pojęć z C++ z krótkimi objaśnieniami.]

3 W j. ang. switch.
4 W j. ang. case.

5.4 Środowisko

5.4.1 Język [Programowania - przyp. JMJ]

Wymaganie Specjalne 8.: Zabrania się stosowania roz. j. Należy stosować się ściśle do standardu C++ [ISO/IEC 14882:2002(E)].

Komentarz 8.: To dobra zasada - pomijając nr normy. Natomiast GNU wmawia, że ich rozszerzenia składni C++ nie są niczym złym. Jasne jest, że to robienie wody z mózgu, bo taki kod nie jest przenośny.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.4.2 Zestaw znaków

[Ten roz. rozpoczyna się uwagą, że jego zawartość się zmieni gdy jeden lub więcej obcych języków będzie używanych na wej. lub wyj. (np. na wyświetlaczach pilota). - przyp. JMJ]

Wymaganie Zwykłe 9.: Dopuszcza się w kodzie jedynie 96 znaków wyspecyfikowanych jako poprawne w standardzie C++.

Komentarz 9.: To dobra zasada - jednak to zaszłość specyficzna dla C++. Ten Zestaw znaków jest narzucany przez standard C++. Tak więc jego przestrzeganie dotyczy kompilatora a nie samego programisty. Tak więc pod tym względem jest to zbędna zasada.

Jest to zaszłość bo w j. D i Pytonie nazwy f., typów i zmiennych mogą być złożone ze znaków Unicode.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 10.: Dopuszcza się w napisach jedynie znaki zdefiniowane przez normę ISO-10646-1.

Komentarz 10.: To dobra zasada. Najprawdopodobniej ta norma określa kodowania UTF czyli Unikodu. Stosowanie jej to dobra praktyka, bo zapewnia bezproblemową obsługę dowolnego ziemskiego języka narodowego.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 11. i 12.: Zabrania się stosowania 2 i 3 literowych substytutów znaków[5].

Komentarz 11. i 12.: To dobra zasada - jednak bez znaczenia. : 2 i 3 literowe substytuty znaków wynikały z tego, że niektóre terminale miały nienormalne klawiatury (bez niektórych klawiszy). Nie wiem gdzie stosowano te wynalazki w dodatku do programowania w C++.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Zwykłe 13.: Zabronione jest stosowanie wielobajtowych znaków i "szerokich znaków".

Komentarz 13.: Jest to zła zasada - bo to zaostrzenie zasady 10. Jest to błędne, bo psuje to co daje dziś Unikod. Ta zasada to jakiś nacjonalistyczny nonsens.

"Szerokie znaki" to też znaki wielobajtowe.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 14.: Wymaga się by sufiksy[6] literowe były wyłącznie wielkoliterowe.

Komentarz 14.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Specjalne 15.: Wymaga się spr. poprawności zasobów w czasie działania programu - programowanie defensywne.

Komentarz 15.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

5.5 Biblioteki

Wymaganie Specjalne 16.: Dopuszczalne są wyłącznie biblioteki z certyfikatami "DO-178B level A" lub "SEAL 1".

Komentarz 16.: Oczywiście, nawet do celów cywilnych do użycia powinny być dopuszczalne wyłącznie certyfikowane programy i biblioteki.

Oprócz tego, że programy były by certyfikowane skończyło by się szaleństwo zarzucania rynku co chwilę nowymi programami różniącymi się tylko nr wersji.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania.

5.5.1 Standardowe Biblioteki

Wymaganie Specjalne 17.: Zabronione jest użycie errno. Ale gdy nie można inaczej i jest to dobrze zdefiniowane i udokumentowane to można używać errno (jest tak np. w

5 W j. ang. digraphs i trigraphs.

6 prefiks: to inaczej przedrostek; sufiks: to inaczej przyrostek. Np. AAA_nazwa_BBB - tu AAA to prefiks, a BBB to sufiks.

bibliotekach matematycznych dostarczanych przez firmy trzecie).

Komentarz 17.: To zła zasada - przeczy sobie.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 18.: Zabronione jest Użycie makra offsetof() z stddef.h.

Komentarz 18.: To dobra zasada. Makro offsetof() zwraca indeks w bajtach zmiennej w strukturze. Wygląda na to, że w C++ jego zachowanie jest niezdefiniowane. Nie wiem kiedy można by zastosować makro offsetof() .

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 19.: Zabronione jest użycie f. setlocale() z locale.h.

Komentarz 19.: To zła zasada - bo to znaczy, że wszystko musi być W j. ang. - jakiś nacjonalistyczny nonsens.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 20.: Zabronione jest użycie makr setjmp() i longjmp() z setjmp.h .

Komentarz 20.: To dobra zasada. setjmp() i longjmp() to makra do wykonywania "długich skoków". Normalnie nie są potrzebne. Jeszcze nigdy nie musiałem z nich korzystać.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 21.: Zabroniona jest obsługa sygnałów sys. Uniks za pomocą f. z signal.h.

Komentarz 21.: To dobra zasada. Sygnały sys. Uniks to b. dziwna komunikacja z programem.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 22.: Zabronione jest użycie f. z stdio.h.

Komentarz 22.: To dobra zasada. W stdio.h są f. w j. C do obsługi strumieni operujących na plikach. W C++ są zupełnie inne strumienie.

Strumienie z C++ też są do niczego, bo się nie nadają do napisów z uwagi na nie możliwość ich tłumaczenia gdy się przeplata tekst z danymi.

Np.:

```
cout << _("To jest: ") << 100 << _(" ale więcej jest: ") << 102 << endl;
```

Tego nie da się prawidłowo przetłumaczyć, bo napisy nie są parametryzowane tylko poszatkowane.

Prawidłowo jest to w Qt:

```
QTextStream(stdout) << QObject::tr("To jest %1 ale więcej jest %2").arg(100).arg(102) << endl;
```

Są niezależne biblioteki w C++ do parametryzowania napisów, np. świetna biblioteka fmt .

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 23.: Zabronione jest użycie f. atof() , atoi() i atol() z stdlib.h . Ale gdy jest to dobrze zdefiniowane i udokumentowane to można ich używać.

Komentarz 23.: To zła zasada - przeczy sobie.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 24.: Zabronione jest użycie f. abort() , exit() , getenv() i system() z stdlib.h .

Komentarz 24.: Dziwi mnie zakaz stosowania f. exit() , bo jednak są sytuacje gdy program nie może kontynuować działania.

Dziwi mnie zakaz stosowania f. getenv() z uwagi na to, że ja osobiście mam wrażenie, że przekazywanie do programu parametrów przez zmienne środowiskowe jest uważane za "bardziej profesjonalne" - GitLab w skryptach yaml do budowania paczek używa wyłącznie zmiennych środowiskowych i wcale nie można przekazywać ze skryptu yaml do swoich skryptów parametrów linii komend. Dlatego jestem zdania, że w projekcie F-35 cała konfiguracja mieści się w plikach tekstowych jakie są wciągane wraz z uruchomieniem każdego programu.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 25.: Zabronione jest użycie f. z time.h .

Komentarz 25.: To dziwna zasada. Widać, że mają ją przykrytą jakimś własnym interfejsem. Ale o tym tu nie wspominają.

Ocena: Wada.

Zastosowanie: BRAK.

5.6 Dyrektywy Peprocesora

Wymaganie Specjalne 26.: Dopuszczalne są wyłącznie poniższe dyrektywy preprocesora:

```
#ifndef
#define
#endif
#include
```

Komentarz 26.: To zła zasada. Pow. dyrektywy za mało do normalnego programowania w C++ ani nawet w C. Trzeba ją uzupełnić o dyrektywy:

```
__PRETTY_FUNCTION__ na sys. Uniks lub
__FUNCTION__ na sys. Windows
__FILE__
__LINE__
```

Są one konieczne przy identyfikacji miejsc wykrycia błędów w kodzie: rzucanie wyjątków, zapis logów, wypisywanie błędów na stderr.

Dodatkowo sklejanie napisów jest konieczne by sklejać napisy w makrodefinicjach. Ja robię to by zrobić taki trik:

```
#define mMakroDoNapisuPomocnicze2(a,
aNapis) a##aNapis
#define mMakroDoNapisuPomocnicze1(aNapis)
mMakroDoNapisuPomocnicze2(L, #aNapis)
#define mMakroDoNapisu(aNapis)
mMakroDoNapisuPomocnicze1(aNapis)
```

mMakroDoNapisu() konwertuje zawartość makrodefinicji do napisu Unicode w UTF-32 (kompilator GNU), lub UTF-16 (kompilator M$). Czyli:

```
#define mNazwaFirmy Energo Kod
std::wstring
lNazwaFirmy(mMakroDoNapisu(mNazwaFirmy));
```

Efektywnie daje to taki kod do kompilacji:

```
std::wstring lNazwaFirmy(L"Energo Kod");
```

To bezproblemowo można konwertować do np. do QString f. QString::fromStdWString(). Sprytne nie?!?

Robię nawet lepszy numer: W pliku projektu CMakeList.txt definiuję sobie makra jakie mają wjechać do programu przy kompilacji. Dzięki temu i pow. mMakroDoNapisu() wszystkie metadane projektu mam w pliku projektu a nie w źródłach. Mówię tu o takich metadanych: ORGANIZACJA, AUTORZY, NAZWA_PROJEKTU, WERSJA i inne.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 27.: Dopuszczalne są wyłącznie dyrektywy #ifndef , #define , #endif do zapobiegania wielokrotnemu włączaniu plików nagłówkowych.

Przykład:

```
#ifndef Header_filename
#define Header_filename

    // Header declarations...

#endif
```

Komentarz 27.: Patrz Komentarz 26.. Na tamtym etapie nie było innego roz. Natomiast od C++2020 można stosować moduły, które rozwiązują ten problem całkowicie.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Zwykłe 28.: Dopuszczalne jest użycie dyrektyw #ifndef i #endif wyłącznie takie jak w zasadzie 27.

Komentarz 28.: To dobra zasada - ale może być lepsza: kodując w Polsce należy ten przykład spolszczyć np. tak:

```
#ifndef Naglowek_NazwaPliku
#define Naglowek_NazwaPliku

    // Deklaracje...

#endif // Naglowek_NazwaPliku
```

Ja dodaję po #endif komentarz taki jak w dyrektywie #ifndef i #define pow.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 29.: Zabronione jest użycie makra #define do tworzenia f. wstawianej[7].

Uzasadnienie:

1. "Inline functions do not require text substitutions" - po 25 latach programowania w C++ nie wiem co to znaczy.
2. F. wstawiane mają norm. kontrolę typów.

Komentarz 29.: To zła zasada. Przy tym ograniczeniu nie da się wszystkiego dobrze zakodować. Np. do rzucania wyjątków potrzebne jest takie makro pomocnicze:

```
#define mWyjatek(aKomponent, aKod,
aTresc) \

enpro::Wyjatek(enpro::Wyjatek::eException
, aKomponent, aKod, aTresc \
      , Napis::fromUtf8(__FILE__),
__LINE__, Napis::fromUtf8(mNazwaFunkcji))
```

W makro mWyjatek() chodzi o to, żeby mieć prawidłowe wart.: nazwa pliku, funkcja i linia. Bez takiego makra było by to niemożliwe, chyba że język miałby jakieś inne mechanizmy takie jak np. zrzut stosu znany z Pytona.

C++ nie ma możliwości zrzutu zawartości stosu wywołań f. w razie awarii programu.

Ocena: Partactwo.

---

7 W j. ang. inline.

Zastosowanie: BRAK.

Wymaganie Specjalne 30.: Zabronione jest stosowanie #define do definiowania stałych. Do definiowania stałych należy używać dyrektywy stała[8].

Komentarz 30.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 31.: Dopuszczalne jest użycie dyrektywy #define wyłącznie w celu zapobiegania wielokrotnym włączeniom plików nagłówkowych.

Komentarz 31.: To zła zasada. W pow. przykładach podałem użycie dyrektyw #define których ani dyrektywy stała ani szablon nie zastąpi.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 32.: Dopuszczalne jest użycie dyrektywy #include wyłącznie do włączania plików nagłówkowych.

Komentarz 32.: To dobra zasada - jednak to truizm.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.7 Pliki Nagłówkowe

Wymaganie Specjalne 33.: Zabronione użycie włączeń lokalnych z apostrofami. Dopuszczalne jest wyłącznie użycie dyrektywy #include w charakterze włączeń systemowych, czyli z nawiasami trójkątnymi.

Przykłady:

```
#include <foo.h>                // Dobrze.
#include <dir1/dir2/foo.h>      // Dobrze: Ścieżka względna.
#include “foo.h”                // Źle: włączenie lokalne.
```

Komentarz 33.: To zła zasada - nonsens. Jak są kompilatory które włączenia obsługują niestandardowo, to należy je porzucić jako nienormalne (bo szaleństwo wciąga jak wir). "due to the unfortunate divergence in vendor implementations" - to żaden argument, bo zasady są jasne:

1. #include <PlikNagluwkowy.h++> // Włączenie nagłówka systemowego, czyli z systemu operacyjnego;
2. #include "PlikNagłuwkowy.h" // Włączenie nagłówka z bieżącego projektu.

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 34.: Dopuszczalne w nagłówkach są jedynie związane ze sobą deklaracje.

Uzasadnienie: Ogranicza to ilość zbędnych powiązań.

Komentarz 34.: To dobra zasada - jednak to truizm.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Zwykłe 35.: Wymagane jest zabezpieczenie w każdym pliku nagłówkowym, by nie włączał on wielokrotnie sam siebie.

Komentarz 35.: To dobra zasada - jednak to truizm: łamiąc tą zasadę nie da się programować w C++.

Ocena: Zaleta.

Zastosowanie: BRAK.

Zalecenie 36.: Ograniczaj ilość zależności wymagających rekompilacji.

Komentarz 36.: To dobra zasada - jednak to truizm.

UWAGA: Jak dotąd nikt nie wspomniał o prekompilowanych nagłówkach, a dopiero to daje znaczące przyspieszenie kompilacji.

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 37.: Dopuszczalne jest włączanie do pliku nagłówkowego wyłącznie tego co konieczne do jego kompilacji. To czego brakuje w nagłówku do kompilacji pliku *.c++ musi być włączane w pliku *.c++ .

Uzasadnienie: Zmniejsza to zależności w kodzie.

Komentarz 37.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Zalecenie 38.: Zabronione jest włączanie do nagłówków plików z deklaracjami kl. do których odwołania odbywają się przez wsk. i ref.

Uzasadnienie: Zmniejsza to zależności w kodzie.

Komentarz 38.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 39.: Zabronione jest umieszczanie w pliku nagłówkowym definicji nie stałych zmiennych oraz definicji f.

8 W j. ang. const.

Uzasadnienie: Plik nagłówkowy musi zawierać jedynie deklaracje, bez szczegółów implementacji.

Komentarz 39.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.8 Pliki Definicji

Wymaganie Specjalne 40.: Wymaga się by nazwy f. wstaw., typy i szablony miały różne nazwy.

Komentarz 40.: To zła zasada - tego pilnuje kompilator.

Ocena: Partactwo.

Zastosowanie: BRAK.

5.9 Styl

Zasady formatowania kodu mają tą zaletę, że czynią kod łatwiejszym do zrozumienia. Zawsze kiedy to możliwe generatory kodu należy skonfigurować zgodnie z poniższymi zasadami.

Wymaganie Zwykłe 41.: Zabrania się w kodzie linii dłuższych niż 120 znaków.

Uzasadnienie: Długie linie trudno się czyta.

Komentarz 41.: To dobra zasada - ale może być lepsza: Zabrania się w kodzie linii dłuższych niż 80 znaków.

Uzasadnienie:

1. Można wtedy wyświetlić 2 pliki obok siebie. B. często znacznie ułatwia to pracę z komputerem. Nawet teraz mam wyświetlone obok siebie 2 dokumenty: orginał "F-35 - Coding Rules.pdf" i ten dokument w którym go komentuję;
2. Można wygodnie wyświetlić kody źródłowe nawet na starych komputerach z konsolą 80x25 (standard z M$ DOS).

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

Wymaganie Zwykłe 42.: Wymaga się by każde wyrażenie było w osobnej linii.

Uzasadnienie: Zwiększenie prostoty, czytelności i poprawa stylu.

Komentarz 42.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

Zalecenie 43.: Zabrania się stosowania znaków tabulacji poziomej czyli znaku o kodzie 0x09 w tab. ASCII.

Komentarz 43.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

Wymaganie Zwykłe 44.: Wymaga się by wcięcia miały co najmniej 2 spacje.

Uzasadnienie: Zwiększenie czytelności i poprawa stylu.

Komentarz 44.: To zła zasada: Dawno temu jak się uczyłem C++ przeczytałem w jakiejś książce[9], że wcięcia muszą mieć 3 spacje, bo jak jest ich mniej, to kod jest nieczytelny, a jak jest więcej, to jest to przesada. Ja uważam, że to ma sens.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

5.9.1 Nazewnictwo

Nazwy w kodzie powinny:

1. Sugerować użycie;
2. Powinny być krótkie ale ekspresywne;
3. Powinny być wystarczająco długie by unikać konfliktów nazw. Jednak nie mogą być nadmiernie długie;
4. Powinny zawierać powszechnie znane skróty.

Komentarz: To dobra zasada.

UWAGA: Dopuszczalne są jednoliterowe nazwy jeśli występują w b. małym zakresie widoczności i poprawiają Poprawa czytelności kodu.

UWAGA: Poniżej "słowo" znaczy: słowo. akronim[10] lub skrót[11].

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Zwykłe 45.: Wymaga się by wszystkie słowa w nazwie były rozdzielone znakiem podłogi czyli znakiem

9 Może to była "C++ Księga Experta" wyd. Helion.

10 Akronim to poł. pierwszych liter kilku wyrazowej nazwy. Np: KGHM - to Kombinat Górniczo Hutniczy Miedzi.

11 Skrót to początkowy fragment dłuższego określenia. Np. "spr." to skrót od "sprawdzenie".

'_' (bez apostrofu), czyli znakiem o kodzie 0x5F w tab. ASCII.

Uzasadnienie: Zwiększenie czytelności i poprawa stylu.

Komentarz 45.: To zła zasada - jest to po prostu b. niewygodne.

Uzasadnienie: Można inaczej: Znacznie nowocześniejszym nazewnictwem posługuje się Qt.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 46.: Zabrania się w programie nazw dłuższych niż 64 znaków.

Uzasadnienie: Standard C++ gwarantuje, że kompilator akceptuje nazwy o dł. min. 1024 znaków.

Komentarz 46.: To dobra zasada. Najwidoczniej takie swawole jak nazwy o długości do 1024 znaków wydały się komuś zbytnim luksusem.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 47.: Zabronione jest rozp. nazwy od podłogi czyli znaku '_' (bez apostrofów), czyli znaku 0x5F w tab. ASCII.

Komentarz 47.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 48.: Nazwa nie zmieni się przez:

Zbieg okoliczności;

1. Obecność lub brak podłogi czyli znaku '_' (bez apostrofów);
2. Zamianę znaków 'O' z '0' lub 'D';
3. Zamianę znaków 'I' z '1' lub 'l';
4. Zamianę znaków 'S' z '5';
5. Zamianę znaków 'Z' z '2';
6. Zamianę znaków 'n' z 'h'.

Uzasadnienie: Czytelność.

Komentarz 48.: To dobra zasada, jednak tylko automat może tego pilnować.

Takie zasady jak ta należy kontrować wyłącznie automatycznie - skryptami.

Ocena: Rewelacja.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 49.: Wymaga się by akronim w nazwie składał się wyłącznie z wielkich liter.

Uzasadnienie: Czytelność.

Komentarz 49.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.9.1.1 Nazewnictwo Kl., Struktur, Wyliczeń i Użycie typedef

Wymaganie Zwykłe 50.: Wymaga się by pierwsze słowo w nazwie zaczynało się wielką literą. Natomiast wszystkie pozostałe słowa w nazwie zaczynają się małą literą.

Uzasadnienie: Poprawa stylu.

Przykłady:

```
class Diagonal_matrix { ... };
enum RGB_colors {red, green, blue};
```

Wyjątek: Naśladowanie biblioteki standardowej gdy zastępuje się typy podstawowe.

```
typename C::value_type s=0;
```

Komentarz 50.: To dobra zasada - jednak tylko dlatego że jest, a nie za to jaka jest.

Uzasadnienie: Można inaczej: Znacznie nowocześniejszym nazewnictwem posługuje się Qt.

Ocena: Wada.

Zastosowanie: Brak.

5.9.1.2 Nazewnictwo F., Zmiennych i Parametrów

Wymaganie Zwykłe 51.: Dopuszczalne w nazwach f. i zmiennych są jedynie małe litery.

Uzasadnienie: Poprawa stylu.

Przykład:

```
class Example_class_name
{
public:
   uint16 example_function_name (void);
private:
   uint16 example_variable_name;
};
```

Komentarz 51.: Patrz Komentarz 50..

Ocena: Wada.

Zastosowanie: Brak.

5.9.1.3 Nazewnictwo Stałych i Wyliczeń

Wymaganie Specjalne 52.: Dopuszczalne w nazwach stałych i wyliczeń są jedynie małe litery.

Uzasadnienie: Mimo, że w różnych bibliotekach bywają używane wielkie litery w nazwach stałych i wyliczeń, to można je przesłaniać makrami.

Komentarz 52.: Patrz Komentarz 50.. Ciekawe jak chcą te stałe i wyliczenia przesłaniać makrami skoro zabraniają używania #define ? Patrz Wymaganie Zwykłe 31..

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.9.2 Nazewnictwo Plików

Nazewnictwo plików podlega tym samy zasadom co Nazewnictwo z pewnymi uzupełnieniami.

Wymaganie Zwykłe 53.: Wymaga się by plik nagłówkowy w j. C++ zawsze miał roz. ".h" (bez cudzysłowu).

Komentarz 53.: To zła zasada: kody programu w j. C++ powinny mieć inne roz. niż pliki z kodami w j. C. Często stosuje się w nazewnictwie plików nagłówkowych C++ roz. "*.h" i inne elem. z j. C, one powodują, że programista C++ degeneruje program do rozwiązań z j. C. Dlatego niedopuszczalne jest stosowanie czegokolwiek z j. C: ani zmiennych, ani struktur, ani nazewnictwa plików - wszystko należy przedefiniować w terminologi proj w j. C++. W szczególności pliki nagłówkowe powinny mieć roz. *.h++. Przestrzegania tego należy pilnować skryptem.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 53.1.: Zabronione jest stosowanie w nazwach plików znaków: [‘\"] (bez nawiasów kwadratowych) oraz sekwencji: "/*" i "//" (bez cudzysłowu).

Komentarz 53.1.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 54.: Wymaga się by plik definicji w j. C++ zawsze miał roz. ".cpp" (bez cudzysłowu).

Komentarz 54.: Zła zasada: język nie jest "cpp", tylko "C++"! Więc pliki definicji f. powinny mieć roz. "c++" lub "C++". Ja doszedłem do tego ok. 2020r. i nie napotkałem na żadne trudności w stosowaniu roz. plików "h++" i "c++".

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 55.: Wymaga się by nazwa pliku nagłówkowego logicznie podpowiadała zawartość pliku.

Komentarz 55.: To dobra zasada - jednak to truizm: każdy plik powinien być tak nazwany - nie tylko kody źródłowe.

Ocena: Zaleta - tylko dlatego, że uczniowie mogą tego nie być świadomi.

Zastosowanie: Do zapamiętania.

Zalecenie 56.: Wymaga się by nazwa pliku definicji logicznie podpowiadała zawartość pliku. Normalnie nazwa musi być identyczna z odpowiadającym mu plikiem nagłówkowym (z różnicą w roz.).

W przypadku gdy jednemu plikowi nagłówkowemu odpowiada wiele plików definicji, należy dodawać odpowiednie sufiksy do nazwy pliku.

Przykład 56.1.: Jeden plik definicji do kl. Matrix:

```
Matrix.cpp
```

Przykład 56.2.: Wiele plików definicji dla biblioteki f. matematycznych:

```
Math_sqrt.cpp
Math_sin.cpp
Math_cos.cpp
```

Komentarz 56.: To dobra zasada - jednak może być lepsza: W przypadku nieprzenośnego kodu należy stosować sufiksy wskazujące na dany system operacyjny.

Przykład 56.3.: Definicja f. kl. Plik dla systemu Linuks i Widnows:

```
Plik.linuks.c++
Plik.okna.c++
```

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.9.3 Kl.

Wymaganie Zwykłe 57.: Wymaga się by sekcje w kl. występowały w kolejności: public, protected, private.

Uzasadnienie: Użytkownika kl. najbardziej interesują f. public. Tego kto chce rozszerzać kl. najbardziej interesują f. public i protected. A f. private nikogo nie powinny interesować.

Komentarz 57.: To dobra zasada - ale może być lepsza: Można zauważyć, że f. czyść() oraz destruktor są ostatnimi f. wywoływanymi przez użytkownika kl. Tak więc powinny być na końcu deklaracji kl. To jest zdrowa symbolika: konstruktor zaczyna, potem używa się f. robocze, a na koniec czyści się dane lub niszczy o. - to normalne, prawda?

Ocena: Zaleta.

Zastosowanie: Skrypt.

5.9.4 Funkcje

Wymaganie Zwykłe 58.: Wymaga się by 1. param. f. był w 1. linii, a pozostałe osobno w kolejnych liniach.

Uzasadnienie: Zwiększenie czytelności i poprawa stylu.

Komentarz 58.: To zła zasada. Patrz Komentarz 41.

Ocena: Partactwo.

Zastosowanie: Brak.

5.9.5 Bloki

Wymaganie Specjalne 59.: Wymaga się by po dyrektywach: "if", "else if", "else", "while" (bez cudzysłowów) zawsze występowały nawiasy klamrowe "{}" (bez cudzysłowu). Nawet gdy są to puste bloki.

Uzasadnienie: Trudno zauważyć ';' (bez apostrofu).

Komentarz 59.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 60.: Wymaga się by nawiasy klamrowe "{}" (bez cudzysłowu) występowały w tej samej kolumnie.

Przykład:

```
if(lLiczba == lWart)
{
}
else
{
}
```

Komentarz 60.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

Wymaganie Zwykłe 61.: Zabronione jest dodawanie czegokolwiek do linii z nawiasami klamrowymi.

Wyjątek: komentarze (gdy są konieczne).

Komentarz 61.: To dobra zasada - ale wyjątek zbędny.

Ocena: Zaleta.

Zastosowanie: Skrypt.

5.9.6 Wsk. i Ref.

Wymaganie Zwykłe 62.: Wymaga się by w deklaracji zmiennej znak gwiazdki, czyli '*' (bez cudzysłowu), czyli 0x2A w tab. ASCII, był zapisywany zaraz po typie.

Uzasadnienie: Jest to zapis przekładający nacisk na typ zamiast nacisku na składnię.

Przykłady:

```
int32* p;         // Prawidłowo.
int32 *p;         // Nieprawidłowo.
int32* p, q;      // Prawdopodobnie błąd.
```

Komentarz 62.: To dobra zasada.

Uzasadnienie: Od jakichś 20 lat głoszona jest propaganda, że wsk. w C++ to nie typ. I właśnie 3 linia z pow. przykładu jest tego dowodem, bo zmienna q będzie typu int32 a nie wsk.em typu int32*. Jednak nikt kto zna Asembler nie da się na to nabrać.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

5.10 Klasy

5.10.1 Różne

Wymaganie Zwykłe 63.: Zabrania się stosowania znaku spacji, czyli ' ', czyli 0x20 w tab. ASCII, przed operatorami dostępu "kropka", czyli '.' i "strzałka", czyli "->" (bez apostrofów i bez cudzysłowu).

Komentarz 63.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt formatujący np. używający prog. astyle.

Zalecenie 64.: Wymaga się by interfejsy kl. były kompletne i minimalne.

Uzasadnienie: [Tu podają frazesy - przyp. JMJ]

Komentarz 64.: To dobra zasada. Jest to inna postać motta znanego z kultury systemu Uniks: "Niech to będzie to proste jak to możliwe, ale nie prostsze.".

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.10.2 Rozważania Dotyczące Praw Dostępu

Ogólnie są tylko 2 typy kl.: 1. Do przechowywania danych, 2. Dostarczających abstrakcji przy posiadaniu dobrze zdefiniowanego stanu lub niezmiennika. Poniżej przedstawiono zasady to uwzględniające:

Zalecenie 65.: Wymaga się by struktury składały się wyłącznie z danych nie ograniczanych żadnym niezmiennikiem.

Komentarz 65.: To dziwna zasada, bo każda zmienna w programie musi mieć jakiś nie zmiennk.

Ocena: Wada.

Zastosowanie: Brak.

Zalecenie 66.: Wymaga się by kl. składały się z danych podlegających ograniczeniu definiującym niezmiennik klasy.

Komentarz 66.: To dobra zasada. Te klasy mogłyby się nazywać klasa typu usługa.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

Zalecenie 67.: Zabrania się użycia zmiennych publicznych i poufnych[12] w kl.

Uzasadnienie: Jeśli zmienne są poza sekcją prywatną[13] kl. nie może ich kontrolować.

Wyjątek: Sekcje zastrzeżone mogą występować w przypadku gdy kl. nie dostarcza interfejsu użytkownikom [Programistom używającym biblioteki? - pyt. JMJ].

Komentarz 67.: To zła zasada. Wynika to z faktu, że autor czy dostawca kl. nie jest w stanie przewidzieć do czego będzie musiał ją dostosować programista który będzie jej używał. W programowaniu podobnie jak na froncie - dzieją się rzeczy niepojęte. Dlatego potrzebne są sprytne i wszechstronne narzędzia jak szwajcarski scyzoryk, a nie sam mały kordzik jaki dawali w HJ.

Ocena: Zaleta, bo zakazuje niekontrolowanego dost. do zm. Wada bo zabrania dost. do zm. poufnych z kl. bazowej.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

5.10.3 Funkcje [kl. i struktur - przyp. JMJ]

Wymaganie Specjalne 68.: Zabronione jest dodawanie do kl. i struktur zbędnych niejawnie generowanych f.

Komentarz 68.: To dobra zasada.

Ja mam zasadę, że kod generuję wyłącznie pod nadzorem i włączam go do repo tak jak by był programowany ręcznie. Wyjątkiem są f. gen. przez kompilator z szablonów z j. C++.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.10.4 Funkcje Typu Stała

Wymaganie Zwykłe 69.: Wymaga się deklaracji f. w kl. jako f. stała jeśli nie modyfikuje danych swojej kl.

Uzasadnienie: Oznaczenie f. jako stała powoduje, że nie nastąpi przypadkowa modyfikacja kl.

W standardzie C++ f. deklarowane jako stała może być przeciążona przez identyczną f. bez etykiety stała.

Komentarz 69.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.10.5 Przyjaciele [kl. - przyp. JMJ]

Wymaganie Zwykłe 70.: Dopuszcza się deklarację przyjaciela kl. wyłącznie gdy przyjaciel wymaga dostępu do prywatnych f. i zmiennych kl. Dopuszczalne jest to wyłącznie gdy nie można przyjaciela umieścić w kl. z powodów logicznych lub wydajnościowych.

Uzasadnienie: Nadmierne użycie przyjaciół komplikuje kod i utrudnia jego utrzymanie.

W zasadzie 70. w Dodatku A podany jest przypadek gdy przyjaciele kl. są dopuszczalni. W tych przypadkach przyjaciel kl. jest nie tylko dopuszczalnym rozwiązaniem ale wręcz zalecanym.

UWAGA: Zamiast dodawania przyjaciół można upubliczniać f lub zmienne kl.

Komentarz 70.: To zła zasada. Przyjaciele kl. to czyste zło.

Ocena: Wada.

Zastosowanie: BRAK.

5.10.6 Czas Życia Obiektu, Konstruktory i Destruktory

5.10.6.1 Czas Życia Obiektu

Programista i w sumie każdy myślący człowiek wie, że by bezpiecznie użyć obiektu należ go najpierw stworzyć, natomiast po jego zniszczeniu nie należy go używać. Jednak są przypadki gdy nie jest oczywiste kiedy obiekty są tworzone i niszczone. Stąd poniższe zalecenia.

Wymaganie Specjalne 70.1.: Zabronione jest niewłaściwie [? - przyp. JMJ] użycie obiektu przed i po jego czasie życia.

Uzasadnienie: Takie użycie jest niezdefiniowane.

Komentarz 70.1.: To dobra zasada - choć to truizm dla programisty C++.

UWAGA: W tej zasadzie pomylono nr.

Ocena: Wada.

Zastosowanie: Do zapamiętania - na pocz. nauki C++.

12 W j. ang. protected.
13 W j. ang. private.

Wymaganie Specjalne 71.: Zabronione są wywołania f. jakie udostępnia kl. przed pełnym zainicjowaniem obiektu.

Uzasadnienie: Zapobiega to częściowemu przypisaniu wartości do obiektu.

Komentarz 71.: To dobra zasada. Zastanawiałem się chwilę nad tym i działa to chyba tak:

```
class k1
{
public:
    K1::K1()
    {
        cIstnieje = true;
        sprNiezmiennik();
    }

    void f1()
    {
        [...]
        sprNiezmiennik();
    }

protected:
    void sprNiezmiennik()
    {
        if(!cIstnieje)
            throw std::exception("Obiekt
kl. K1 jeszcze nie został utw.");
        [...]
    }

public:
    K1::~K1()
    {
        cIstnieje = false;
    }

protected:
    bool cIstnieje = false;
};
```

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 71.1.: Zabrania się wywoływania f. wirtualnych w konstruktorach i destruktorach.

Uzasadnienie: W konstruktorach i destruktorach są wywoływane wyłącznie f. kl. do której one należą. Czyli w konstruktorach i destruktorach nie działają f. wirt.

Komentarz 71.1.: To dobra zasada.

To twórcy standardu C++ wymyślili, że f. wirt. nie działają w kostruktorach i destruktorachi. Jest to nienormalne, bo w D i w Pytonie f. wirt. działają w konstruktorach i destruktorach.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Zalecenie 72.: Wymaga się by niezmiennik kl. był sprawdzany:

1. W konstruktorze - na końcu;
2. W f. publicznych kl. - na pocz. i na końcu;
3. W destruktorze - na pocz.

Komentarz 72.: To dobra zasada. Ale może być lepsza:

Wymaga się by niezmiennik kl. był sprawdzany:

1. W konstruktorze - na końcu;
2. W f. publicznych kl. - na końcu.

Nie ma potrzeby spr. niezmiennika na pocz. f. pub. skoro był on spr. na jej końcu. W destruktorze też nie ma sensu jego spr.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 73.: Zabronione jest dodawanie do kl. zbędnych domyślnych konstruktorów.

Uzasadnienie: Użytkownik [Programista korzystający z biblioteki - przyp. JMJ] nie będzie tworzył zbędnych obiektów do puki nie będzie to logicznie uzasadnione.

Komentarz 73.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 74.: Wymaga się by przyp. wart. do zmiennych w kl. było realizowane przez listę inicjalizacyjną, a nie w ciele konstruktora [Po dwukropku, nie w nawiasach klamrowych - przyp. JMJ].

Wyjątek: Stałe wart. zdefiniowane w kl.

Wyjątek: Gdy się nie da, bo wymagane jest dodatkowe przetwarzanie.

Komentarz 74.: To dobra zasada - jednak częściowo nie aktualna: Od implementacji standardu C++ 2011, można przyp. wart. zmiennych w deklaracji kl - jest to dużo czytelniejsze. Jednak w ten sposób nie można przypisać zmiennym kl. wart. param. konstruktora.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Specjalne 75.: Wymaga się by zmienne na inicjalizacyjnej w def. konstruktora były w tej samej kolejności co w deklaracji kl.

Uzasadnienie: Wart. są przyp. w kol. deklaracji zmiennych w kl., a nie kol. na liście przyp. wart.

Komentarz 75.: To dobra zasada. Jednak ma ona niewielkie znaczenie, bo jej nie przestrzeganie generuje

ostrzeżenie kompilatora. Bo kompilator ostrzega o tej różnicy między deklaracją i listą przypisującą wart.

Ocena: Wada.

Zastosowanie: Przegląd kodu z próbą kompilacji w celu spr. ostrzeżeń i błędów kompilatora.

Wymaganie Specjalne 76.: Wymaga się dodania do kl. konstruktora kopiującego i operatora przyp., czyli "operator=()" (bez cudzysłowu) gdy kl. zawierają wsk. lub nietrywialne destruktory.

UWAGA: Zobacz Wymaganie Zwykłe 80. które wskazuje, że konstruktor kopiujący i operator przyp. są zalecane gdy upraszczają kod.

Uzasadnienie: Zapewnienie prawidłowego zarządzania zasobami gdy są one kopiowane.

Komentarz 76.: To dobra zasada. Praktycznie wszystkie kl. należy wyposażać w konstruktor kopiujący i operator przyp. Ja zawsze wywołuję w konstruktorze kopiującym operator przyp.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 77.: Wymaga się by konstruktor kopiujący przyp. wart. wszystkich zmiennych jakie definiują niezmiennik kl., w tym zmienne z kl. bazowych. Dane niezwiązane z niezmiennikiem kl., np. bufory nie muszą być kopiowane.

UWAGA: Jeśli stosuje się zliczanie odwołań nie trzeba kopiować wszystkiego za każdym razem.

Uzasadnienie: Zapewnienie, że o. jest prawidłowo kopiowany. Zobacz: Zasada 77. w Dodatku A.

Komentarz 77.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt ze spr. czy na koniec konstruktora kopiującego wywoływana jest f. sprNiezmiennik.

Wymaganie Specjalne 77.1.: Zabronione jest podawanie w deklaracji konstruktora domyślnych wart. param. które powodują, że jego wywołanie pasuje do konstruktora kopiującego.

Uzasadnienie: Kompilatory nie muszą rozróżniać tej niejednoznaczności. Zobacz: Zasada 77.1. w Dodatku A.

Komentarz 77.1.: To zła zasada - coś takiego nigdy nie powinno się skompilować.

Uzasadnienie: Miałem problem ze zrozumieniem tej zasady. Ale chodzi o takie coś:

```
Kl.
{
public:
   Kl.(Kl.& b);   // Może być generowany
niejawnie.
   Kl.(Kl.& b, int pParam = 1);
};
```

I wywołanie:

```
Kl. a;
Kl. c(a);
```

Ocena: Wada.

Zastosowanie: BRAK.

5.10.6.2 Destruktory

Wymaganie Specjalne 78.: Wymaga się dodania do kl. wirt. destruktora gdy kl. ma f. wirt.

Uzasadnienie: Usuwanie wsk. do obiektu kl. z f. wirt. i bez wirt. destruktora za pośrednictwem wks. do obiektu kl. przodka jest nie zdefiniowane.

Komentarz 78.: To dobra zasada.

To twórcy standardu C++ zmuszają programistów do ręcznego pilnowania, czy destruktor jest wirtualny, czy nie. Mimo, że nic złego się nie dzieje gdy wszystkie destruktory są wirt.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 79.: Wymaga się by wszystkie zasoby jakie uzyskuje kl. były zwalniane przez jej destruktor.

Komentarz 79.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.10.7 Operatory Przyp.

Wymaganie Zwykłe 80.: Wymaga się użycia domyślnego konstruktora kopiującego i domyślnego operatora przyp. "operator=()" (bez podwójnego apostrofu) gdy te oferują sensowną składnię [Kodu - przyp. JMJ].

Uzasadnienie: Domyślne wersje konstruktora kopiującego i operatora przyp. są bardziej poprawne, łatwiejsze w utrzymaniu i wydajniejsze niż generowane ręcznie.

Komentarz 80.: To zła zasada. Do tego by polegać na domyślnych konstruktorach kopiujących potrzebna jest ściąga pokazująca kiedy one są a kiedy ich nie ma - ja nie mam takiej ściągi i dlatego ja tego nigdy nie wiem. W dodatku nie wiem też kiedy domyślny operator kopiowania jest generowany a kiedy nie (nie przypominam sobie bym kiedykolwiek zdołał go użyć - zawsze kompilator protestował, że sam muszę go zaprogramować).

To twórcy standardu C++ wprowadzili te dziwne reguły i niejasności w generowaniu dom. konstruktorów i operatorów, tak że nigdy nie wiadomo kiedy te f. zostaną wygenerowane automatycznie a kiedy trzeba je ręcznie wklepać.

Ciekawe: Dlaczego miałby być domyślne konstruktory kopiujące wydajniejsze od "generowanych ręcznie"? Przecież f. to f - generowanie z automatu i ręcznie powinny być tak samo dobrze optymalizowane.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 81.: Wymaga się by operator przypisania, czyli "operator=()" (bez cudzysłowu) prawidłowo działał na samym sobie.

Uzasadnienie: Kod:

```
a = a;
```
musi działać prawidłowo. Patrz: Zasada 81. w Dodatku A.

Komentarz 81.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 82.: Wymaga się by operator przypisania, czyli "operator=()" (bez cudzysłowu), zwracał ref. do *this.

Uzasadnienie: Typy wbudowane i biblioteka standardowa C++ zachowują się w ten sposób.

Komentarz 82.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 83.: Wymaga się by operator przypisania "operator=()" (bez cudzysłowu) przyp. wart. wszystkich zmiennych jakie są definiują niezmiennik kl., w tym zmienne z kl. bazowych. Dane niezwiązane z niezmiennikiem kl., np. bufory nie muszą być kopiowane.

UWAGA: Aby zakodować kopiowanie w przenośny sposób, trzeba zapewnić, że:

```
class A {};
class B : public A {};
B x1, x2;
```
działanie:

```
x1 = x2;
x1 = x2;
```
musi dawać ten sam efekt jak

```
x1 = x2;
```
Uzasadnienie: Należy zapewnić, że dane z kl. bazowych są kopiowane prawidłowo. Patrz Wymaganie Specjalne 77.

Komentarz 83.: To dobra zasada. Wydaje się, że jedyną opcją by to zrobić jest kaskada wywołań operatorów przypisania w którym "C::operator=()" wywołuje "B::operator=()", a on wywołuje "A::operator=()" (bez cudzysłowów). Wynika to z faktu, że w konstruktorze nie działają f. wirt. więc normalne rozwiązanie w postaci f. wirt. nie wchodzi w grę.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.10.8 Przeładowywanie Oper.

Wymaganie Zwykłe 84.: Wymaga się ograniczania przeładowywania operatorów.

Uzasadnienie: Niezwykłe lub nie spójne użycie operatorów prowadzi do nieporozumień. Operatory muszą działać zgodnie ze swoim znaczeniem i wg przyjętych w j. C++ konwencji. Np.: "a += b;" musi działać tak jak sekwencja "a = a + b;" (bez cudzysłowów).

Komentarz 84.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Przegląd kodu.

Wymaganie Zwykłe 85.: Wymaga się spójnego kodowania przeciwnych operatorów: pierwszy musi odwracać znaczenie drugiego, a drugi musi odwracać znaczenie pierwszego. Np. "operator==()" i "operator!=()" (bez cudzysłowu).

Uzasadnienie: Gdy dostarcza się "operator==()", programista spodziewa się obecności także "operator!=()" (bez cudzysłowów).

Gdy operatory są spójnie zakodowane utrzymanie kodu staje się łatwiejsze.

Komentarz 85.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt spr. obecność jednego z odwracalnych operatorów i spr. obecność drugiego.

5.10.9 Dziedziczenie

Dziedziczenie jest dopuszczalne gdy wymagany jest wybór f. w trakcie działania programu. Gdy nie jest to konieczne, należy używać szablonów. Szablony maja tą zaletę, że "lepiej się zachowują" i są szybsze niż f. wirt. Proste koncepcje powinny mieć postać typów prostych. Jednak wybór konkretnego rozwiązania zależy od złożoności problemu.

Poniższe zasady dostarczają dalszych szczegółów jak powinny być projektowane hierarchie kl.

Zalecenie 86.: Wymaga się by typy proste reprezentowały proste koncepcje.

Uzasadnienie: Dobrze zaprojektowane typy proste zazwyczaj są: są zrozumiałe, efektywne pod względem wielkości i czasu, minimalnie są zależne od innych kl. i możliwe do oddzielnego użycia.

Komentarz 86.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Zalecenie 87.: Wymaga się by hierarchia kl. bazowała na kl. abs.

Uzasadnienie: Kl. abs. zapewnia: "czysty interfejs", oddziela interfejs od szczegółów implementacji, pozwala na jednoczesne istnienie różnych implementacji i ogranicza zależności w trakcie kompilacji.

Komentarz 87.: To dobra zasada. Ale przydatna tylko przy tworzeniu wtyczek.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Specjalne 88.: Wymaga się by dziedziczenie wielobazowe polegało na dziedziczeniu N interfejsów abs. + M implementacji. N interfejsów abs. jest dziedziczonych publicznie. Z M implementacji co najmniej M-1 jest dziedziczona prywatnie. Co najwyżej 1 implementacja jest dziedziczona jako poufna[14].

Uzasadnienie: Dziedziczenie wielobazowe może doprowadzić do skomplikowanej hierarchii kl., co jest trudne do zrozumienia i utrzymania.

Komentarz 88.: To dobra zasada.

Aby zaimplementować tą zasadę kl. dziedzicząca wielobazowo musi przeładowywać f. N interfejsów funkcjami które będą korzystać f. z M kl. implementacji.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 88.1.: Wymaga się by przy dziedziczeniu wielobazowym wszystkie kl. w hierarchii dziedziczyły wirt. po bazowej kl. abs.

Uzasadnienie: Jawne zdefiniowanie wirt. dziedziczenia mówi, że nie można czynić założeń dotyczących wyłącznego dostępu do danych kl. bazowej. Zobacz: Zasada 88.1 w Dodatku A.

Komentarz 88.1.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 89.: Zabrania się by kl. bazowa (abs.) była jednocześnie wirtualna i niewirtualna w tej samej hierarchii [Kl. - przyp. JMJ].

Komentarz 89.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Zalecenie 90.: Wymaga się by często używane interfejsy były minimalne, ogólne i abstrakcyjne.

Uzasadnienie: Zapewnia to stabilność interfejsów w obliczu zmieniającej się hierarchii kl. dziedziczących.

Komentarz 90.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 91.: Wymaga się by publiczne dziedziczenie kl. B po kl. A oznaczało: "B jest rodzajem A".

Uzasadnienie: Przy publicznym dziedziczeniu kl. A przez kl. B, ta druga jest bardziej wyspecjalizowanym typem A. Czyli "B jest rodzajem A".

Natomiast dziedziczenie prywatne kl. C w kl. D jest kwestią czysto implementacyjną - nie ma bezpośredniego związku między interfejsem kl. bazowej C i interfejsem kl. pochodnej D.

Komentarz 91.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 92.: Wymaga się, przy dziedziczeniu kl. A przez kl. B, aby każda f. wirt. w kl. B miała:

1. Warunki wstępne równe lub słabsze od tych w odpowiadająca jej f. wirt. z kl. A;
2. Warunki końcowe równie lub mocniejsze od tych w odpowiadająca jej f. wirt. z kl. A.

Czyli kl. B będzie wymagała mniej i dostarczała więcej niż kl. A. Ta zasada powoduje, że wszystkie dziedziczone kl. będą zgodne z Zasadą Podstawienia Barbary Liskow.

Uzasadnienie: Tak osiąga się przewidywalne zachowanie kl. dziedziczących (B) występujących w kontekście kl. bazowej (A). Patrz: Zasada 92 w Dodatku A.

Komentarz 92.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

14 W j. ang. protected.

Wymaganie Zwykłe 93.: Wymaga się by powiązania typu: "posiada" oraz "dziedziczenie implementacyjne" były realizowane przez zmienne kl. lub przez dziedziczenie prywatne.

Uzasadnienie: Dziedziczenie publiczne oznacza "B jest rodzajem A". Patrz: Zasada 93 w Dodatku A.

Komentarz 93.: Takie dziedziczenie jest bez sensu.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 94.: Zabronione jest przedefiniowanie zwykłych f. (nie wirt.).

Uzasadnienie: Daje to zabezpieczenie przed niespójnym zachowaniem, gdy:

```
class A
{
   void f();
};

class B :  public A
{
   void f();
};
```

To prawdopodobne jest wywołanie raz "A::f()" a innym razem "B::f()", wtedy gdy oczekuje się wywołania "B::f()" (bez cudzysłowów) .

Komentarz 94.: To dobra zasada.

Jednak dostawcy bibliotek nie deklarują wszystkich publicznych f. jako wirt. A to jest sprzeczne z elementarną zasadą programowania obiektowego: "Kl. zamykam przed modyfikowaniem i otwieram na rozszerzanie.".

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 95.: Zabroniona jest zmiana domyślnej wart. parametru f. wirt.

Uzasadnienie: Przedefiniowanie domyślnego param. f. wirt. często prowadzi do niespodziewanych rezultatów.

Komentarz 95.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 96.: Zabronione jest użycie polimorfizmu w przypadku prostych tablic.

Uzasadnienie: W C++ dostęp do prostych tablic realizowany jest przez arytmetykę wsk. Czyli:

```
Kl. a[100];
a[i]; // znaczy tyle co:
a + i * SIZEOF(Kl.);
```

Często kl. pochodna jest większa niż kl. bazowa, wiec arytmetyka wsk. nie będzie działać poprawnie w kl. pochodnej.

Komentarz 96.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: BRAK.

Wymaganie Specjalne 97.: Zabronione jest stosowanie prostych tablic. Zamiast nich należy stosować kl. Array.

Uzasadnienie: Gdy proste tablice przekazuje się jako parametry f. to degenerują się one do wsk.

UWAGA: Zobacz Array.doc[15] w celu zapoznania się z poprawnym użyciem kl. Array.

Komentarz 97.: To dobra zasada - zaleca użycie jakiegoś kontenera zamiast prostych tablic.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.10.10 Funkcje Wirtualne

Wymaganie Specjalne 97.1.: Zabronione jest tworzenie operatorów "operator==()" i "operator!=()" (bez cudzysłowów) jako wsk. do f. wirt.

Uzasadnienie: Zachowanie takich operatorów jest niezdefiniowane.

Kilka dodatkowych zasad dotyczy f. wirt. i polimorfizmu. Wymieńmy je tu by były dostępne z jednego miejsca: 71, 78, 87, 97, 221.

Komentarz 97.1.: To dobra zasada, jednak już nieaktualna. Miała ona znaczenie w czasach gdy w C++ nie można było dziedziczyć operatorów. Nie jest dla mnie jasne która wer. standardu to zmienia, ale na 100% w C++2020 mogą być wirt. operatory działają normalnie. Więc nie ma już potrzeby deklarowania operatora wywołującego f. wirt.

Ocena: Zaleta.

Zastosowanie: BRAK.

5.11 Przestrzenie Nazw

Zalecenie 98.: Wymaga się umieszczania wszystkich nazw w jakiejś przestrzeni nazw.

Wyjątek: Lokalne nazwy.

Wyjątek: F. "main" (bez cudzysłowu).

Uzasadnienie: Zabezpiecza to przed konfliktami w dużych programach.

15 Niestety tym dokumentem nie dysponuję - przyp. JMJ.

Komentarz 98.: Jest to zła zasada - w tym zakresie stosuję inną:

Wszystkie f. i kl. z biblioteki muszą się znajdować w przestrzeni nazw. Uważam, że konflikty w obrębie programu nie powinny występować.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Zwykłe 99.: Zabrania się by przestrzenie nazw były zagnieżdżone na więcej niż 2 poziomy.

Uzasadnienie: To zapewnia prostotę i przejrzystość. Głęboko zagnieżdżone przestrzenie nazw są trudne do zrozumienia i poprawnego użycia.

Komentarz 99.: To zła zasada.

Nie ma sensu zawierać przestrzeni nazw w innej przestrzeni nazw. Tak samo nonsensem jest deklarowanie kl. w kl. lub deklarowanie f. w f. Nawet stosowanie f. lambda znane z C++2011 zaciemnia dramatycznie kod (i sam osobiście ich nie stosuję).

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 100.: Wymaga się by dostęp do elementów w przestrzeni nazw odbywał się na jeden z trzech sposobów:

1. Gdy używa się nie wielu elementów z przestrzeni nazw (sposób 1.):

```
namespace N
{
class A {...};
}

using N::A;                    // Deklaracja
użycia, lub:
```

2. Gdy używa się nie wielu elementów z przestrzeni nazw (sposób 2.):

```
N::A;                          // Jawne podanie
przestrzeni nazw
```

3. Gdy się używa wielu elementów z przestrzeni nazw:

```
using namespace N;             // Dyrektywa
użycia
```

Uzasadnienie: Gdy używa się tylko kilku el. z danej przestrzeni nazw nie trzeba zaciągać wszystkiego.

Komentarz 100.: To dobra zasada. Nawet nie wiedziałem o deklaracji użycia.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.12 Szablony

Szablony to potężny mechanizm pozwalający generować f. i kl. parametryzując je typami. Dzięki temu uzyskujemy f. i kl. jakie pod względem wielkości i szybkości działania są takie same jak programowane ręcznie.

Choć szablony są potężną techniką programowania, to może wydawać się nie jasne kiedy użyć szablonu zamiast dziedziczenia. Następujące wskazówki dostarczył Pan Stroustrup:

1. Stosuj szablony zamiast dziedziczenia gdy wydajność jest priorytetem;
2. Stosuj dziedziczenie zamiast szablonów gdy dodawanie nowych modułów bez przebudowania jest priorytetem[16];
3. Stosuj szablony zamiast dziedziczenia gdy nie ma wspólnego przodka;
4. Stosuj szablony zamiast dziedziczenia gdy typy wbudowane i struktury wymuszają zgodność.

Wymaganie Specjalne 101.: Obowiązują nast. zasady przeglądu szablonów:

1. Spr. założeń lub wymagań dla parametru szablonu - tych jakie przewidywał autor;
2. Spr. jakie f. zostały wygenerowane z szablonu przez kompilator na podst. aktualnych param.

UWAGA: Kompilator tak należy skonfigurować, by podwał listę wygenerowanych f. z szablonów. Zobacz: Zasada 101 w Dodatku A.

Uzasadnienie: Ponieważ wiele f. może być wygenerowane z szablonu, recenzent kodu musi spr. co już zostało wygenerowane i jakie są założenia i wymagania dla argumentów danego szablonu.

Komentarz 101.: To dobra zasada - ale tylko w projektach rządowych - cywilne projekty nie maja na to środków.

Ocena: Zaleta.

Zastosowanie: Skrypt podający listę wygenerowanych f. wirt.

Wymaganie Specjalne 102.: Wymaga się by testy szablonu pokrywały wszystkie f. aktualnie wygenerowane z szablonu.

UWAGA: Kompilator tak należy skonfigurować, by podwał listę wygenerowanych f. z szablonów. Zobacz: Zasada 102 w Dodatku A.

16 Czyli gdy program oparty jest na wtyczkach - przyp. JMJ.

Uzasadnienie: Z szablonu może być wygenerowane wiele f. Dlatego wszystkie należy przetestować.

Komentarz 102.: To dobra zasada - ale tylko w projektach rządowych - cywilne projekty nie maja na to środków.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Zalecenie 103.: Wymaga się kontroli typów przyjmowanych przez szablony.

Uzasadnienie: Jawna kontrola parametrów daje jasne komunikaty o błędach. Patrz: Zasada 103 w Dodatku A.

Komentarz 103.: To zła zasada. Wyobraźmy sobie kontrolę typów przyjmowanych przez std::vector. To po prostu jest nie realne i przeczy samej idei szablonów.

Ta zasada była by do przyjęcia w sytuacji gdyby obowiązywała w jakichś szczególnych przypadkach. Jednak nie ma tu żadnych takich uwag.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 104.: Wymaga się by specjalizacja szablonu występowała przed jego użyciem.

Uzasadnienie: To wymaganie standardu j. C++.

Przykład:

```
template<class T> class List {...};
List<int32*> li;
      // Użycie specjalizacji szablonu.
template<class T> class List<T*> {...};
      // BŁĄD: spec. szablonu po użyciu.
```

Komentarz 104.: To zła zasada: skoro to część standardu, to chyba może być kontrolowana przez kompilator?

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 105.: [Niezrozumiałe - przyp. JMJ] "A template definition’s dependence on its instantiation contexts should be minimized.".

Uzasadnienie: [Niezrozumiałe - przyp. JMJ] "Since templates are likely to be instantiated in multiple contexts with different parameter types, any nonlocal dependencies will increase the likelihood that errors or incompatibilities will eventually surface."

Komentarz 105.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 106.: Wymagana jest specjalizacja szablonów dla wsk. (gdy ma to sens).

Uzasadnienie: Wsk. zwykle wymagają specjalnej obsługi.

Komentarz 106.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

5.13 Funkcje

5.13.1 Deklaracje F., Definicje F. Oraz Parametry F.

Wymaganie Specjalne 107.: Zabrania się deklaracji f. w zakresie bloku kodu. Wymaga się by f. były deklarowane w zakresie pliku.

Uzasadnienie: Deklaracje f. w blokach może być mylące.

Komentarz 107.: To dobra zasada. Ja osobiście jestem przeciwnikiem f. lambda - bo one też łamią tą zasadę.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Specjalne 108.: Zabronione są f. ze zmienną l. param.

Uzasadnienie: Trudna kontrola typów.

UWAGA: Czasami przeciążenie f. z innymi parametrami jest zamiennikiem zmiennej l. param. Patrz: Zasada 108 w Dodatku A.

Komentarz 108.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Zalecenie 109.: Wymaga się by f. była zaimplementowana w pliku "*.c++".

Wyjątek: F. wstawiane.

Uzasadnienie: Włączanie kodu do pliku deklaracji utrudnia czytanie kodu. Jedynie krótkie f., które powinny być wstawiane w miejscu wywołania, mogą być umieszczane w pliku "*.h++" z deklaracją kl.

Komentarz 109.: To dobra zasada.

To twórcy standardu j. C++ zmuszają programistów do takich nonsensów jak ręczne specyfikowanie f. wstawianych w celu prymitywnej optymalizacji. Wymuszanie specyfikowania f. wstawianych jest tym bardziej nonsensowne, że nie są znane kryteria kiedy one są wstawiane a kiedy nie (bo dyrektywa inline to jakby prośba a nie wymuszenie wstawienia f.)

Normalnie w darmowych kompilatorach mogło by to być załatwione kilkoma warunkami logicznymi np.

wybierającymi do wstawiania wszystkie f. o dł. <= 12 linii.

Kompilatory komercyjne powinny mieć typ kompilacji "profilowanie" z dokładnym zliczaniem wywołań f. I na podst. tych danych - W PEŁNI AUTOMATYCZNIE POWINNA BYĆ OPTYMALIZOWANA WER. RELEASE.

Tak samo nonsensowne jest ręczne zarządzanie indeksami w j. SQL.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Zwykłe 110.: Zabrania się f. które mają więcej niż 7 param.

Uzasadnienie: F. które nie spełniają tego warunku są trudne do czytania, użycia i utrzymania. Takie f. wskazują na niewystarczające użycie koncepcji obiektowości i abstrakcji.

Komentarz 110.: To dobra zasada - ale mogła by być lepsza i zabraniać f. z więcej niż 5 param.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Specjalne 111.: Zabrania się zwracania wks. lub ref. do obiektów lokalnych.

Uzasadnienie: Takie obiekty nie istnieją po wyjściu z f.

Komentarz 111.: To dobra zasada - to podstawowa zasada w C++.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Zalecenie 112.: Zabrania się zwracania wyników jakie trzeba ręcznie niszczyć operatorem delete.

Uzasadnienie: Może to łatwo doprowadzić do wycieku zasobów. Patrz: Zasada 173. i oraz 112. w Dodatku A.

Komentarz 112.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

5.13.2 Typy i Wart. Zwracane

Wymaganie Zwykłe 113.: Wymaga się by f. miała tylko jeden p. wyjścia.

Uzasadnienie: Wiele p. wyjścia z f. powoduje, że jest ona trudna do zrozumienia.

Wyjątek: Pojedynczy p. wyjścia z f. nie jest wymagany gdy znacząco komplikuje kod.

Komentarz 113.: To zła zasada - przeczy sama sobie.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 114.: Wymaga się by f., które deklarują że zwracają wart., zwracały wynik dyrektywą wynik[17].

Uzasadnienie: Brak dyrektywy wynik w takich f. prowadzi do niezdefiniowanego zachowania.

Komentarz 114.: To dobra zasada.

To twórcy standardu C++ wprowadzili brak błędu kompilacji gdy f. która deklaruje, że zwraca wart. nie robi tego. Prowadzi to do błędów trudnych do zrozumienia.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Zwykłe 115.: Wymaga się spr. kodów błędów zwracanych przez f.

Uzasadnienie: Ignorowanie kodów błędów powoduje, że wcześniejsze założenie o prawidłowym stanie aplikacji przestaje być prawdziwe.

Komentarz 115.: To zła zasada. Wynik f. nigdy nie powinien być kodem błędu. Kody błędów powinny być częścią rzucanych wyjątków. Nawet f. bibl. C powinny być opakowane interfejsami w C++.

Ocena: Partactwo.

Zastosowanie: BRAK.

5.13.3 Parametry F. (Wart., Wsk. Lub Ref.)

Zalecenie 116.: Wymaga się by typ prosty był przekazywany przez wart z f. A do f. B jeśli f. A nie potrzebuje zmienionej wart. tego param[18].

Uzasadnienie: Przekazywanie przez wart. typu prostego jest najprostsze i najbezpieczniejsze.

UWAGA: Typ złożony musi być przekazywany przez wsk., ref., lub być typami polimorficznymi.

Zalecenie 117.: Wymaga się przekazywania param. przez ref. gdy nie może on przyjmować wart. NULL.

Zalecenie 117.1.: Wymaga się przekazywania param. przez stałą ref.[19] gdy nie będzie on ulegał zmianie.

---

17 W j. ang. return.

18 Bo potencjalnie może on ulec zmianie w f. B - przyp. JMJ.

19 W j. ang. const T&.

Zalecenie 117.2.: Wymaga się przekazywania param. przez ref.[20] gdy może on być zmieniany.

Zalecenie 118.: Wymaga się przekazywania param. przez wsk. gdy może on mieć wart. NULL.

Zalecenie 118.1.: Wymaga się przekazywania param. przez stały wsk.[21] gdy nie powinien być modyfikowany.

Zalecenie 118.2.: Wymaga się przekazywania param. przez wsk.[22] gdy może być modyfikowany.

Komentarz 116.-118.2.: To złe zasady.

Uzasadnienie: Należy przyjąć taką zasadę: F. może przyjmować jako param. wyłącznie stałą ref.[23] lub stały wsk.[24]. Aby wyprostować myślenie i znormalnieć należy tak programować f.:

Parametr f. musi być tylko i wyłącznie parametrem,

wystąpienie błędu musi być zgłaszane wyjątkiem,

a wynik f. musi być wyłącznie wynikiem (a nie żadnym błędem) I MUSI BYĆ TYLKO JEDEN!

Uzasadnienie: Tak jest w ziemskiej matematyce: f. zwraca jedną wart.

Uzasadnienie: Stała ref. i stały wsk. ma zawsze ten sam rozmiar 4 lub 8 bajtów (odpowiednio dla proc. 32 i 64 bit.) - tego bardziej zoptymalizować nie można z uwagi na wyrównania danych[25].

Ocena: Partactwo.

Zastosowanie: BRAK.

5.13.4 Wywołania F.

Wymaganie Specjalne 119.: Zabroniona jest rekurencja.

Uzasadnienie: Stos ma ograniczone rozmiary.

Wyjątek: Rekurencja jest dopuszczalna gdy:

1. Wytwarza się SEAL 3 [? - przyp. JMJ] lub oprogramowanie użytkowe.
2. Jest udowodnione, że stos jest wystarczający w przypadku maks. poziomu wywołań rekurencyjnych.

Komentarz 119.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

20 T&.
21 const T*.
22 T*.
23 const T&.
24 const T*.
25 W j. ang. aligment.

5.13.5 Przeładowanie F.

Zalecenie 120.: Wymaga się by przeładowane f. miały takie samo zastosowanie i różniły się wyłącznie param.

Uzasadnienie: Jeśli przeładowane f. służą do różnych celów prowadzi to to zamieszania.

Komentarz 120.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

5.13.6 Funkcje Wstawiane

[Tu opis koncepcji, przykłady i implementacja w j. C++ f. wstawianych]

Zalecenie 121.: Zabrania się f. wstawianych mających więcej niż 2 instrukcje.

Uzasadnienie: Kryteria wstawiania f. są różne w różnych kompilatorach. Patrz: Zasada 121 w Dodatku A.

Komentarz 121.: To dobra zasada - pod warunkiem, że chodzi o 2 linie, a nie "instrukcje", bo instrukcje mamy w j. Asembler.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Zalecenie 122.: Wymaga się by proste f. czytające i ustawiające były wstawiane.

Uzasadnienie: Oszczędza to miejsce i czas.

Komentarz 122.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Zalecenie 123.: Wymaga się ograniczania l. f. czytających i ustawiających w kl.

Uzasadnienie: Duża l. f. czytających i ustawiających wskazuje na to, że kl. pełni rolę zbioru danych zamiast być abstrakcją z dobrze zdefiniowanym stanem lub niezmiennikiem. W tym przypadku lepszym rozwiązaniem może być wydzielona struktura z danymi publicznymi. Zobacz: Zasada 64. i Zasada 66.

Komentarz 123.: To dobra zasada - wręcz ciekawa.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania.

Zalecenie 124.: Wymaga się by f. jedno liniowe były wstawiane.

Uzasadnienie: Oszczędza to miejsce i czas.

Komentarz 124.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

5.13.7 Obiekty Chwilowe

Zalecenie 125.: Zabrania się tworzyć zbędnych obiektów chwilowych.

Uzasadnienie: Wywołania skomplikowanych konstruktorów lub destruktorów spowalnia program.

Komentarz 125.: To dobra zasada - ale to truizm.

Ocena: Wada.

Zastosowanie: Do zapamiętania - na początku nauki C++.

5.14 Komentarze

Komentarze można podzielić na nagłówkowe i towarzyszące kodowi:

1. Komentarze w plikach nagłówkowych są dla tych co używają kl.;
2. Komentarze w plikach z kodem f. są dla tych co modyfikują kl.

Taki podział jest dobry ale tylko w teorii: komentarze w plikach nagłówkowych (*.h++) degenerują te pliki, bo przestają one być czytelne, a to przeczy samej ich koncepcji. Dlatego wszystkie komentarze powinny być w plikach definicji, czyli *.c++.

Komentarze mogą być też podzielone na strategiczne lub taktyczne:

1. Komentarze strategiczne wyjaśniają co f. lub fragment kodu powinien robić. Są umieszczane przed kodem.
2. Komentarze taktyczne wyjaśniają co dana linia kodu robi. Niestety zbyt wiele taktycznych komentarzy czyni kod nieczytelnym. Dlatego komentarze powinny być przede wszystkim strategiczne. Wyjątkiem jest b. skomplikowany kod.

Wymaganie Specjalne 126.: Zabrania są komentarzy w stylu j. C[26]. Prawidłowe są wyłącznie komentarze w stylu C++[27]. Zobacz Zasadę 126 w Dodatku A.

Uzasadnienie: Jeden standard wprowadza spójność w kodzie.

Wyjątek: Generatory kodu które nie mogą być skonfigurowane zgodnie z tą regułą.

26 /* [...] */
27 // [...]

Komentarz 126.: To dobra zasada. Wyjątek zły.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Specjalne 127.: Wymaga się usuwania zakomentowanego kodu.

Uzasadnienie: Zabrania się "martwego kodu".

Komentarz 127.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 128.: Zabronione są komentarze (działań lub źródeł) poza plikiem.

Uzasadnienie: Komentarz wymaga zmian wyłącznie gdy zmienia się plik. [Dalej tekst jest niezrozumiały - przyp. JMJ] "Note that this rule does not preclude the documentation of valid assumptions that may made be entities contained within the file.".

Komentarz 128.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Zalecenie 129.: Wymaga się by komentarz w nagłówku wyjaśniał wyłącznie zachowanie f. z p. widzenia jej użycia.

Uzasadnienie: Omawianie w komentarzu nagłówkowym szczegółów jakie są w kodzie prowadzi do użycia zgodnego z tymi szczegółami.

Komentarz 129.: To dobra zasada.

Uzasadnienie: Wszelkie niestandardowe wymogi f. musi weryfikować sama na samym jej pocz. W życiu nie ma czasu na zawracanie sobie głowy szczegółami implementacji każdej f. z osobna.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Zalecenie 130.: Wymaga się by każda linia kodu była opisana przez komentarz. Jednak jeden komentarz może dotyczyć wielu linii kodu.

Uzasadnienie: Czytelność. [Później powtórzenie treści zalecenia - przyp Jacka Marcina Jaworskiego]

Komentarz 130.: To zła zasada. Kod powinien być samo opisowy. Natomiast komentarze powinny dotyczyć rzeczy kluczowych i potencjalnie niejasnych. Przed deklaracją f.:

1. Co robi f.?
2. Jakie parametry do f. są prawidłowe?
3. Jakie zwraca wyniki?

Natomiast w kodzie (definicji) f. powinny być wyjaśnione stosowane algorytmy.

Uzasadnienie: Zasady retoryki: każdy tekst powinien charakteryzować się ekonomicznością wypowiedzi oraz uwzględniać odbiorcę.

Ocena: Partactwo.

Zastosowanie: Brak.

Zalecenie 131.: Zabrania się komentowania tego co jest jasne w kodzie.

Uzasadnienie: Zbędne komentarze trzeba utrzymywać.

Przykład: Źle skomentowany kod:

```
a = b + c;  // Dodaje b do c i przypisuje do a.
```

Komentarz 131.: To dobra zasada.

Uzasadnienie: Zakłada, że kod może być samo opisowy.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 132.: Wymagany jest komentarz do każdej: deklaracji zmiennej, typedef, wartości wyliczenia, zmiennej w strukturze.

Wyjątek: Gdy komentarz powiela informację z kodu.

Komentarz 132.: To dobra zasada.

Uzasadnienie: Zakłada, że kod może być samo opisowy.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 133.: Wymaga się by każdy plik źródłowy rozpoczynał się komentarzem opisowym zawartości pliku: nazwa pliku, zawartość, ew. noty prawne i prawa autorskie.

Komentarz 133.: To zła zasada.

Uzasadnienie: Nazwa pliku jest znana. Zawartość jest opisana komentarzami przed kl. i przed każdą f. Noty prawne powinny dotyczyć całych pakietów oprogramowania, a nie pojedynczych plików. Noty prawne powinny być wyłączone do osobnych plików w kat. "dok" (bez cudzysłowu), bo nie mają one związku z kodem.

Ocena: Partactwo.

Zastosowanie: BRAK.

Zalecenie 134.: Wymaga się by założenia robione przez f. były wymienione przed deklaracją f.

Uzasadnienie: Gdy te założenia są nieznane, to utrzymanie kodu jest trudne.

Komentarz 134.: To zła zasada - sprzeczna z Zalecenie 129.

Ocena: Partactwo.

Zastosowanie: BRAK.

5.15 Deklaracje i Definicje

Wymaganie Specjalne 135.: Zabrania się przykrywania zmiennych z wyższych zakresów (bloków).

Uzasadnienie: Jest to b. mylące.

Przykład:

```
int32 sum = 0;
{
   int32 sum = 0;
   ...
   sum = f (x);
}
```

Komentarz 135.: To dobra zasada. Gdy się spotyka kod jak w przykładzie prowadzi to prosto do wściekłości.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Zalecenie 136.: Wymaga się by deklaracje kl., f. i zmiennych były umieszczane w najmniejszym możliwym zakresie widoczności.

Uzasadnienie: Należy ograniczać l. zmiennych o których trzeba pamiętać.

[W tym uzasadnieniu pojawia się dodatkowe wymaganie: - przyp. JMJ] Tworzenie zmiennej należy opóźnić do momentu zgromadzenia wszystkich koniecznych danych. Patrz: Zasada 136 Dodatek A.

Komentarz 136.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Zalecenie 137.: Wymaga się by zmienne były statyczne gdy tylko to możliwe.

Uzasadnienie: [Niezrozumiałe - przyp. JMJ] "Minimize dependencies between translation units where possible.". Patrz: Zasada 137 Dodatek A.

Komentarz 137.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 138.: [Niezrozumiałe - przyp. JMJ] "Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.".

Uzasadnienie: Ukrywanie nazw zmiennych może być mylące. Patrz: Zasada 138 w Dodatku A.

Komentarz 138.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 139.: Wymaga się by deklaracja kl. f. i zmiennej występowała wyłącznie raz w plikach programu.

Uzasadnienie: Unikanie sprzecznych deklaracji. Patrz: Zasada 139 Dodatek A.

UWAGA: Tego typu błędy są wykrywane przez linker, lecz później niż jest to oczekiwane. [Reszta niezrozumiała - przyp. JMJ] " (i.e. the inconsistency could exist in a different group's build.) Normally this will mean declaring external objects in header files which will then be included in all other files that need to use those objects (including the files which define them)."

Komentarz 139.: To zła zasada - tego pilnuje linker!

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 140.: Zabronione jest użycie dyrektywy rejestr[28].

Uzasadnienie: Obecne kompilatory potrafią prawidłowo używać rejestrów.

Komentarz 140.: Do dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 141.: Zabronione jest deklarowanie kl., struktury lub wyliczenia w pliku definicji.

Uzasadnienie: Czytelność. Patrz: Zasada 141 w Dodatku A.

Komentarz 141.: To dobra zasada. Ale może być lepsza: kl. (w tym kl. typu struktura) muszą być deklarowane w osobnych plikach *.h++ i implementowane w jednym lub więcej plików *.c++.

Uzasadnienie: Czytelność w sensie łatwości analizy projektu.

Ocena: Zaleta.

Zastosowanie: Skrypt.

28 W j. ang. register.

5.16 Przypisanie Wartości

Wymaganie Specjalne 142.: Wymagane jest przypisanie wart. każdej zmiennej przed jej użyciem. Patrz zasady: 71, 73, 136, 143.

Uzasadnienie: Zabezpieczenie przed użyciem zmiennej przed poprawnym przypisaniem wartości.

Wyjątek: Wyjątkiem od tej zasady są tam gdzie nazwy[29] są dostępne zanim można przypisać wartość (np. wart. pobrana ze strumienia wej.).

Komentarz 142.: To dobra zasada. Wyjątek moim zdaniem zbędny - nie znam takiego przypadku.

Ocena: Zaleta.

Zastosowanie: Skrypt (a raczej flaga kompilatora).

Wymaganie Zwykłe 143.: Zabrania się wprowadzania zmiennych bez prawidłowych wartości. Patrz zasady: 73, 136, 142.

Uzasadnienie: Zabezpieczenie przed udostępnianiem zmiennych z błędnymi wartościami.

Komentarz 143.: To dobra zasada - jednak to rozszerzenie zasady 142.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 144.: Wymaga się stosowania nawiasów klamrowych[30] przy przypisaniu wartości strukturom, wtedy gdy wart. są niezerowe. Dotyczy to również tablic.

Uzasadnienie: Czytelność.

Przykład:

```
int32 a[2][2] = { {0,1} ,{2,3} };
```

Komentarz 144.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 145.: Wymaga się zdefiniowania wszystkich wart. wyliczenia gdy definiuje się nowe wyliczenie.

Wyjątek: Pierwsza wart. może być nadana automatycznie.

Uzasadnienie: Mieszanie wart. wyliczeń podanych ręcznie i automatycznie jest b. mylące.

Komentarz 145.: To dobra zasada - jednak może być lepsza i pomijać wyjątek.

29 Zmiennych? - przyp. JMJ.

30 {}

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

5.17 Typy

Wymaganie Specjalne 146.: Wymagana jest obsługa l. zmiennoprzecinkowych zgodnie ze standardem ANSI/IEEE Std 754.

Uzasadnienie: Zachowanie spójności.

Komentarz 146.: To dobra zasada - Tylko pojawia się pyt.: Jakie inne implementacje są spotykane w C++? Podejrzewam, że chodzi o procki z poza firm AMD lub Intel.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 147.: Zabrania się manipulacji l. zmiennoprzecinkowymi na poziomie bitów.

Uzasadnienie: Powoduje to liczne błędy. Patrz: Zasada 147 z Dodatku A.

Komentarz 147.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania.

Wymaganie Specjalne 148.: Wymaga się by każdy zbiór wart. do wyboru był ujęty w wyliczenie.

UWAGA: [Niezrozumiała - przyp. JMJ] "This rule is not intended to exclude character constants (e.g. ‘A’, ‘B’, ‘C’, etc.) from use as case labels.".

Uzasadnienie: Poprawa uruchamiania ze śledzeniem, czytelności i łatwości utrzymania kodu.

UWAGA: Jeśli to możliwe, powinno się włączyć flagę ostrzegania o braku wszystkich wart. wyliczenia w dyrektywie dopasuj[31].

Komentarz 148.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.18 Stałe

Roz. 4.6.2 zawiera dodatkowe uwagi na temat stałych, wyliczeń i makr.

Wymaganie Specjalne 149.: Zabrania się użycia stałych w systemie ósemkowym.

Wyjątek: Zero.

Uzasadnienie: L. zaczynające się od zero '0' (bez cudzysłowu) są w C++ traktowane jako stałe w systemie ósemkowym. W celu unikania pomyłek zmiennych w systemie ósemkowym należy unikać.

UWAGA: Stałe w systemie szesnastkowym są dopuszczalne.

Komentarz 149.: To dobra zasada - jednak z innego powodu: ósemkowy system l. jest rzadko stosowany i przez to mylący.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 150.: Wymaga się by zmienne w systemie szesnastkowym były zapisywane wyłącznie cyframi i wielkimi literami.

Komentarz 150.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 151.: Zabrania się używania w kodzie "liczb magicznych", zamiast nich będą używane symbole.

Uzasadnienie: Poprawa czytelności i łatwości utrzymania kodu.

Wyjątek: Przypisywanie wart. do tablic wart. l.

```
class A
{
   A()
   {
      coefficient[0] = 1.23; // Dobrze
      coefficient[1] = 2.34; // Dobrze
      coefficient[2] = 3.45; // Dobrze
   }
private:
   float64 coefficient[3]; // Jej wart.
nie może być przypisana przez listę
inicjalizacyjną.
};
```

Wyjątek: L. 1 i 0 można używać gdy są one wart. logicznymi lub gdy 0 reprezentuje NULL.

Komentarz 151.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 151.1.: Zabrania się modyfikowania napisów [W stylu j. C - przyp. JMJ].

UWAGA: Jest to zabronione standardem, jednak wiele kompilatorów tego nie przestrzega.

Uzasadnienie: Próba zmodyfikowania napisu jest niezdefiniowana.

Komentarz 151.1.: To dobra zasada.

31 W j. ang. switch.

Ocena: Zaleta.

Zastosowanie: Flaga kompilatora lub skrypt.

5.19 Zmienne

Wymaganie Specjalne 152.: Zabrania się deklaracji więcej niż jednej zmiennej w linii.

Uzasadnienie: Poprawa czytelności kodu i eliminacja niespodzianek. Patrz: Zasada 62.

Przykład:

```
int32* p, q; // Prawdopodobnie błąd.
int32 first
button_on_top_of_the_left_box, i; // Źle:
można przeoczyć zmienną 'i' (bez
cudzysłowu).
```

Komentarz 152.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

5.20 Unie i Pola Bitowe

Wymaganie Specjalne 153.: Zabrania się używania unii.

Uzasadnienie: Statyczna kontrola typów w przypadku unii nie istnieje. Historycznie wiadomo, że unie były powodem błędów.

Komentarz 153.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 154.: Wymaga się by pola bitowe były wyłącznie l. bez znaku lub wyliczeniami.

Uzasadnienie: To czy pola bitowe zdefiniowane jako char, short, int, long są ze znakiem czy nie, to zależy od implementacji. Dlatego jawne zadeklarowanie, że nie mają one znaku zapobiega rezerwacji miejsca na znak oraz zapobiega przekręcaniu licznika.

UWAGA: [Niezrozumiała - przyp. JMJ] "MISRA Rule 112 no longer applies since it discusses a two-bit minimum-length requirement for bit-fields of signed types.".

Komentarz 154.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 155.: Zabrania się używania pól bitowych w celu oszczędzania pam.

UWAGA: Pola bitowe powinny być używane do komunikacji ze sprzętem i w protokołach sieciowych.

UWAGA: Pewne aspekty manipulacji polami bitowymi są zależne od implementacji.

Uzasadnienie: Pola bitowe oznaczają konieczność wygenerowania dodatkowego kodu wymaganego do manipulacji nimi. Ten kod jest wolniejszy niż normalne f. operujące na l.[32].

Komentarz 155.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 156.: Wymaga się by dostęp do zmiennych kl. odbywał się za pośrednictwem ich nazw.

Uzasadnienie: Zapis i odczyt pam. za pomocą samych wsk. generuje błędy.

Wyjątek: Pola bitowe zerowej długości mogą być użyte do określenia wyrównania w pam. następnego pola bitowego[33].

Komentarz 156.: To dobra zasada - jednak tylko dlatego, że w standardzie C++ nie ma właściwości - jednak już w drugiej poł. lat 90. XXw. właściwości były dostępne w pakiecie w Borland C++ Builder.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu; Skrypt.

5.21 Operatory

Wymaganie Specjalne 157.: Zabrania się "specjalnych efektów" prawej strony wyrażenia "&&" lub "||" (bez cudzysłowu).

Uzasadnienie: Czytelność. Gdy prawa strona wyrażenia generuje "specjalne efekty" jest to łatwe do przeoczenia. Patrz: Zasada 157 w Dodatku A.

Komentarz 157.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 158.: Wymaga się stosowania nawiasów okrągłych[34] w przypadkach gdy w wyrażeniu z "&&" lub "||" (bez cudzysłowów) występują dodatkowe operatory logiczne.

Uzasadnienie: Czytelność. Patrz: Zasada 158 w Dodatku A.

---

32 Na całe szczęście ten kod jest generowany automatycznie - przyp. JMJ.

33 Wynika to z faktu większej efektywności przesyłania danych w większych porcjach w procesorach 32bit. jakie pojawiły się w latach 1980. To samo rozwiązanie stosuje się w prockach 64bit. - przyp. JMJ.

34 ().

Komentarz 158.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 159.: Zabronione jest przeciążanie operatorów "&&", "||" (bez cudzysłowu) i '|' (bez apostrofu).

Uzasadnienie:

1. Wbudowane operatory "&&", "||" (bez cudzysłowu) i te same operatory ale przeładowane inaczej się zachowują. Może to prowadzić do nieoczekiwanych rezultatów.
2. Jeżeli zostanie przekazany adres do kl. która została jedynie zapowiedziana, czyli bez pełnej deklaracji, i ta kl. definiuje operator& , to zachowanie jest niezdefiniowane.

Komentarz 159.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Skrypt.

Wymaganie Specjalne 160.: Wymaga się by operator przypisania był używany wyłącznie do przypisania wart.

Uzasadnienie: Czytelność. operator= może być łatwo pomylony z operator==. Patrz: Zasada 160 w Dodatku A.

Komentarz 160.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 162.: Zabrania się mieszania w wyrażeniach l. ze znakiem i bez.

Uzasadnienie: To prowadziło by do błędów.

Komentarz 162.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Flaga kompilatora.

Wymaganie Specjalne 163.: Zabronione są obliczenia na l. bez znaku.

Uzasadnienie: Używanie l. bez znaku prowadzi do ich mieszania z l. ze znakiem. To prowadzi do naruszenia zasady 162.

Komentarz 163.: To zła zasada. Gdy l. nie ma znaku uzyskuje się 2x większe wart. maks. Ma to znaczenie np. przy maks. ilości el. w kontenerach.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 164.: Wymaga się by operator przesunięcia bitowego w lewo[35] przyjmował wart. >= 0 i wart. < dł. (w bitach) zmiennej na której działa.

Uzasadnienie: Zachowanie dla innych wart. b jest niezdefiniowane.

Komentarz 164.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 164.1.: Wymaga się by operator przesunięcia bitowego w prawo[36] działał na l. >= 0.

Uzasadnienie: Dla e1 >> e2; Gdy e1 jest l. ze znakiem, wtedy zachowanie operatora przesunięcia bitowego w prawo nie jest ujęte w standardzie C++ (jest zdefiniowane przez dostawcę kompilatora).

Komentarz 164.1.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do Zapamiętania; przegląd kodu.

Wymaganie Specjalne 165.: Zabrania się odejmowania l. bez znaku.

Komentarz 165.: To zła zasada. W kontenerach i do operacji na dużych zakresach danych konieczne jest używanie l. bez znaku.

Ocena: Partactwo.

Zastosowanie: Brak.

Wymaganie Zwykłe 166.: [Niezrozumiałe - przyp. JMJ] "The sizeof operator will not be used on expressions that contain side effects."

Uzasadnienie: [Niezrozumiałe - przyp. JMJ] "Clarity. The side-effect will not be realized since sizeof only operates on the type of an expression: the expression itself will not be evaluated."

Komentarz 166.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 167.: Wymaga się rozpoznania i udokumentowania alg. dzielenia typu int jakie zaimplementowano w kompilatorze.

Uzasadnienie: Jeśli jeden lub wiele argumentów dzielenia są ujemne, to wynik tej operacji jest zdefiniowany przez dostawcę kompilatora.

Komentarz 167.: To dobra zasada.

---

35 operator<<().
36 operator>>().

Ocena: Rewelacja.

Zastosowanie: Testy prog. narzędziowych przed rozp. proj.

Wymaganie Specjalne 168.: Zabrania się używania operatora przecinka.

Uzasadnienie: Czytelność. Patrz Zasada 168. w Dodatku A.

Komentarz 168.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

5.22 Wsk. i Ref.

Zalecenie 169.: Zabrania się użycia wsk. do wsk.

Uzasadnienie: W celu unikania błędów należy używać kontenerów lub innych form abstrakcji.

Komentarz 169.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 170.: [Niezrozumiałe - przyp. JMJ] "More than 2 levels of pointer indirection shall not be used."

Uzasadnienie: [Niezrozumiałe - przyp. JMJ] "Multiple levels of pointer indirections typically produce code that is difficult to read, understand and maintain."

UWAGA: [Niezrozumiałe - przyp. JMJ] "This rule leaves no room for using more than 2 levels of pointer indirection. The word “shall” replaces the word “should” in MISRA Rule 102."

Komentarz 170.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 171.: [Niezrozumiałe - przyp. JMJ] "Relational operators shall not be applied to pointer types except where both operands are of the same type and point to:

• the same object,

• the same function,

• members of the same object, or

• elements of the same array (including one past the end of the same array).

Note that if either operand is null, then both shall be null. Also, “members of the same object” should not be construed to include base class subobjects (See also AV Rule 210)."

Uzasadnienie: Łamanie pow. zasady może prowadzić do nieoczekiwanego działania.

Komentarz 171.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 173.: Adres zapisany w sprytnym wsk. nie może być przypisany do zmiennej, która trwa dłużej niż ten sprytny wsk.

Uzasadnienie: Obiekt pod sprytnym wsk. jest niszczony po likwidacji sprytnego wsk.

Komentarz 173.: To zła zasada.

Uzasadnienie: Ta zasada powinna brzmieć: Zabronione jest przypisanie wsk. zmiennej nie będącej sprytnym wsk.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 174.: Zabrania się dereferencji wsk. NULL.

Uzasadnienie: Jest to niezdefiniowane.

Komentarz 174.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 175.: Zabrania się porównywania wks. z wart NULL. Zamiast NULL należy używać po prostu '0' (bez cudzysłowu). To samo dotyczy przypisywania wart. do wsk.

Uzasadnienie: Makro NULL jest zdefiniowane przez dostawcę kompilatora (np. jako '0', "0L", "(void*)0" (bez cudzysłowu)).

Komentarz 175.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 176.: Wymaga się przezywania dyrektywą typedef wsk. do f.

Uzasadnienie: Poprawa czytelności kodu. Wsk. do f. negatywnie wpływają na czytelność kodu.

Komentarz 176.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Przegląd kodu.

5.23 Konwersje Typów

Zalecenie 177.: Zabrania się stosowania operatorów konwersji typów.

Uzasadnienie: Takie operatory mogą być użyte niejawnie, gdy jest to nieoczekiwane. Patrz: Zasada 177 z Dodatku A.

Komentarz 177.: To zła zasada. Programista musi wiedzieć co robi.

W standardzie C++ jest inny problem: "przekleństwo ambigous" które powoduje, że gdy istnieje więcej niż jeden sposób dopasowania parametrów f., to kompilator odmawia posłuszeństwa. Jest tak nawet gdy istnieje dopasowanie dokładne. Jest to całkowicie nienormalne.

Natomiast j. D zachowuje się prawidłowo, czyli normalnie: automatyczne dopasowanie musi być dokładne lub dotyczyć potomków typu do którego się dopasowuje. Pozostałe rodzaje dopasowania wymagają operatorów rzutowania - i jasne jest, że ma to sens jedynie przy rzutowaniu w dół, czyli do potomków kl. bazowej. Pozostałe przypadki dopasowywania i rzutowania są bez sensu.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 178.: Zabrania się rzutowania w dół (do kl. potomnych).

Wyjątek 1.: Użycie f. wirtualnych. Są one przydatne w prostych przypadkach.

Wyjątek 2.: Użycie wzorca wizytator (lub podobnego). Jest on przydatny w bardziej skomplikowanych przypadkach.

Uzasadnienie: Rzutowanie do kl. potomnych jest niebezpieczne. Dlatego potrzebny jest jakiś dodatkowy mechanizm zabezpieczający.

Uwaga 1.: Zabronione jest użycie pól typu, bo prowadzą one do zbyt wielu błędów.

Uwaga 2.: Zabronione jest użycie dyrektywy "dynamic_cast" (bez cudzysłowu) z uwagi na brak wsparcia w narzędziach[37], jednak w przyszłości mogą one być rozważone jeśli odpowiedni przegląd/badanie będzie przeprowadzone dla oprogramowania spełniającego wymagania SEAL 1/2. "dynamic_cast" (bez cudzysłowu) jest w dobry w programowaniu aplikacji ogólnego przeznaczenia[38].

Komentarz 178.: To zła zasada. dynamic_cast działa!

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 179.: Zabrania się konwersji wsk. do kl. bazowej do wsk. kl. pochodnej.

Uzasadnienie: Dysponując wsk. do kl. bazowej nie można prawidłowo rzutować do kl. pochodnej z uwagi na to, że w czasie kompilacji nie można ustalić rozmiaru typu pochodnego. Jest to nie możliwe bez użycia "dynamic_cast" (bez cudzysłowu) lub stosowania f. wirt.

Komentarz 179.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Wsk. do kl. bazowej nie może być konwertowany do kl. pochodnej.

Wymaganie Specjalne 180.: Zabronione są niejawne konwersje jakie mogą prowadzić do utraty danych.

Uzasadnienie: Programista może być nieświadomy, że następuje utrata danych. Patrz: Zasada 180 w Dodatku A.

UWAGA 1.: Można użyć szablonów do rozwiązywania problemów z konwersją typów.

UWAGA 2.: W kompilatorze należy włączyć flagę ostrzegającą o konwersji z utratą danych.

Komentarz 180.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Flaga kompilatora.

Wymaganie Zwykłe 181.: Zabronione jest wielokrotne rzutowanie.

Uzasadnienie: Jest to zbędne zaśmiecanie kodu i generuje problemy gdy później nastąpi zmiana typu zmiennej.

Komentarz 181.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 182.: Zabronione jest użycie rzutowania z "void" (bez cudzysłowu) do i ze wskaźników.

Uzasadnienie: Nie jest to zdefiniowane standardem C++ - definiuje to dostawca kompilatora. Może nastąpić obcięcie wsk. jeśli konwertuje się go do "void" (bez cudzysłowu) w przypadku gdy jest on mniejszy od wsk.

Wyjątek 1.: Dopuszczalne jest rzutowanie z "void*" do "T*" (bez cudzysłowów). Wtedy wymaga się pewności, że "void*" na prawdę jest "T*" oraz wymaga się użycia dyrektywy "static_cast" (bez cudzysłowów). Tego typu kod może występować jedynie w niskopoziomowych f. zarządzania pam.

37 Do uruchamiania ze śledzeniem?!? - przyp. JMJ.
38 Cywilnego?!? - przyp. JMJ.

Wyjątek 2.: Dopuszczalne jest jawne przypisywanie adresów do wks.

```
Device_register input =
reinterpret_cast<Device_register>(0XFFA);
```

Komentarz 182.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Zalecenie 183.: Zabronione jest rzutowanie typów.

Uzasadnienie: Błędy wynikające z rzutowania są najbardziej dokuczliwe częściowo dlatego, że są trudne do wykrycia. Stosuj ścisłe typowanie.

Komentarz 183.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 184.: Zabrania się konwersji l. rze. do całk.

Wyjątek 184.1.: Gdy wymaga tego algorytm.

Wyjątek 184.2.: Gdy wymaga tego sprzęt.

Uzasadnienie: Konwersja l. rze. do całk. powoduje przekręcenie licznika oraz utratę precyzji.

Wyjątek 184.3.: Dopuszczalna jest jawna konwersja l. rze. do całk. dla wykonania operacji matematycznych (przy uwzględnieniu wpływu na szybkość i możliwość przekręcenia licznika). Jeśli to konieczne należy wyraźnie podać w jakiej sytuacji przekręcenie licznika nie następuje.

Komentarz 184.: To zła zasada - przeczy sobie.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 185.: Dopuszczalne jest wyłącznie rzutowanie w stylu C++: "const_cast", "reinterpret_cast" i "static_cast" (bez cudzysłowów).

Uzasadnienie: Rzutowanie w stylu C jest bardziej niebezpieczne niż rzutowanie w stylu C++. Rzutowanie w stylu C jest trudne do znalezienia w dużych programach. Rzutowanie w stylu C może oznaczać konwersję spokrewnionych typów, nie przenośną konwersję między niepowiązanymi typami, lub kombinację konwersji.

Komentarz 185.: To dobra zasada, jednak powinno być dozwolone użycie rzutowanie dynamic_cast – jw.

Ocena: Partactwo.

Zastosowanie: BRAK.

5.24 Pętle i Instrukcje Warunkowe

Wymaganie Specjalne 186.: Zabroniony jest "martwy kod" (który nie jest osiągalny w żadnym przebiegu programu).

UWAGA: Nieużywane f. szablonowe nie będą włączane do plików obiektowych "*.o" (bez cudzysłowu).

Komentarz 186.: To dobra zasada - jednak uwaga jest nie trafiona: nie używane f. szablonowe nie są rozwijane przez kompilator (GNU g++) i nawet nie podlegają kompilacji. Zresztą na logikę: Jak ma być wygenerowana f. z szablonu, gdy nie podano param. szablonu?

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 187.: [Niezrozumiałe - przyp. JMJ] "All non-null statements shall potentially have a side-effect."

Uzasadnienie: "A non-null statement with no potential side-effect typically indicates a programming error. See AV Rule 187 in Appendix A for additional information."

Komentarz 187.: BRAK.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 188.: Zabrania się używania etykiet.

Wyjątek 188.1: Dyrektywa dopasuj[39].

Wyjątek 188.2: Zagnieżdżone pętle.

Uzasadnienie: Poza "dopasuj" etykiety są używane w "idź do"[40].

Komentarz 188.: To dobra zasada - po za wyjątkiem 188.2.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Skrypt.

Wymaganie Specjalne 189.: Zabronione jest użycie dyrektywy "idź do".

Uzasadnienie: Nadmierne użycie "idź do" utrudnia zrozumienie kodu.

Wyjątek: Patrz: Wyjątek 188.2

Komentarz 189.: To dobra zasada - po za wyjątkiem.

Ocena: Do zapamiętania; Skrypt.

Zastosowanie: .

39 W j. ang. switch.
40 W j. ang.: goto.

Wymaganie Specjalne 190.: Zabronione jest użycie dyrektywy "dalej"[41].

Komentarz 190.: To zła zasada: przeczy zasadzie projektowej zakładającej optymistyczny scenariusz w działaniu programu. A wg mnie kod powinien być wiernym odwzorowaniem tej zasady projektowej. W przypadku dyrektywy "dalej" jej stosowanie ma sens dlatego, że na optymistycznej ścieżce programu należy ograniczać ilość warunków oraz ilość sekcji "{}" (bez cudzysłowu) - należy tak robić właśnie dlatego, by optymistyczna ścieżka była jak najprostsza i najczystsza. Wtedy ktoś kto zna dokumentację funkcjonalną, widzi to samo w kodzie.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 191.: Zabronione jest użycie dyrektywy przerwij[42].

Wyjątek: Sekcja dopasuj[43].

Komentarz 191.: To zła zasada - patrz: Komentarz 190.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Zwykłe 192.: Wymaga się by każda dyrektywa spr[44] miała na końcu sekcję przeciw[45].

Uzasadnienie: Strategia defensywna. Patrz: Zasada 192 w Dodatku A.

UWAGA: Tą zasadę stosuje się tylko i wyłącznie gdy po dyrektywie spr[46] następuje jedna lub więcej dyrektyw przeciw spr[47].

Komentarz 192.: To zła zasada, bo sprzeczna z prog. wg optymistycznego scenariusza.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 193.: Wymaga się by każda sekcja wzor[48] kończyła się dyrektywą przerwij[49].

Uzasadnienie: Eliminowanie potencjalnych błędów. Patrz: Zasada 193 w Dodatku A.

41 W j. ang. continue.
42 W j. ang. break.
43 W j. ang. switch.
44 W j. ang. if.
45 W j. ang. else.
46 W j. ang. if.
47 W j. ang. else if.
48 W j. ang. case.
49 W j. ang. break.

Komentarz 193.: To zła zasada. Często się zdarza, że parę wart. w dopasuj[50] powinno być obsłużone jednakowo. Gdy się przestrzega tej zasady prowadzi to do powielania kodu - a to dopiero jest koszmar.

Ocena: Wada.

Zastosowanie: BRAK.

Wymaganie Specjalne 194.: Wymaga się by każda sekcja dopasuj[51] kończyła się dyrektywą domyślne[52].

Wyjątek: Gdy w sekcji wzor[53] występują wszystkie wart. możliwe dla parametru dyrektywy dopasuj[54].

Uzasadnienie: Unikanie ostrzeżeń kompilatora. Brak sekcji domyślne[55] oznacza, że wszystkie możliwe wart. sekcji dopasuj[56] muszą być wymienione w sekcjach wzor[57].

Komentarz 194.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 195.: Zabronione jest użycie logik[58] jako param. sekcji dopasuj[59].

Uzasadnienie 195: Dla testowania wart. logik[60] należy stosować dyrektywę spr[61].

Komentarz 195.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 196.: Każda sekcja dopasuj[62] musi mieć co najmniej 2 sekcje wzor[63] i sekcję domyślne[64].

Uzasadnienie: Uzasadnienie 195:

Komentarz 196.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Specjalne 197.: Zabronione jest używanie liczb rzeczywistych jako liczników pętli.

50 W j. ang. switch.
51 W j. ang. switch.
52 W j. ang. default.
53 W j. ang. case.
54 W j. ang. switch.
55 W j. ang. default.
56 W j. ang. switch.
57 W j. ang. case.
58 W j. ang. bool.
59 W j. ang. switch.
60 W j. ang. bool.
61 W j. ang. if.
62 W j. ang. switch.
63 W j. ang. case.
64 W j. ang. default.

Uzasadnienie: Zaokrąglanie i przycinanie liczb rzeczywistych.

Komentarz 197.: To dobra zasada - jednak nie tylko z tego powodu:

Powinniśmy pamiętać, że pętle w programowaniu odpowiadają szeregom z matematyki, a w nich kol. wart. są numerowane liczbami naturalnymi.

Jest to normalne i uzasadnione przyjętym standardom w ziemskiej matematyce.

Ocena: Zaleta.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 198.: Wymaga się by przypisanie wart. w pętli rob[65] było proste: wart. liczbowa lub poj. wywołanie f. typu daj()[66].

Przykład:

```
for (Iter_type p = c.begin() ; p != c.end() ; ++p)  // Dobrze
{
...
}
```

Uzasadnienie: Poprawa czytelności kodu.

Komentarz 198.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Przegląd kodu. Skrypt.

Wymaganie Zwykłe 199.: Wymaga się by sekcja zwiększania licznika w pętli rob[67] jedynie zwiększała licznik.

Uzasadnienie: Poprawa czytelności kodu.

Komentarz 199.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Skrypt.

Wymaganie Zwykłe 200.: Zabrania się w pętli rob[68]: pustej sekcji przypisania wart, lub pustej sekcji zwiększenia licznika. Zamiast tego należy użyć pętli dopuki[69].

Uzasadnienie: W tym przypadku pętla dopuki[70] wygląda bardziej naturalnie.

Komentarz 200.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Skrypt.

65 W j. ang. for.
66 W j. ang. get
67 W j. ang. for.
68 W j. ang. for.
69 W j. ang. while.
70 W j. ang. while.

Wymaganie Specjalne 201.: Wymaga się by licznik pętli rob[71] był modyfikowany wyłącznie w sekcji zwiększania licznika (a nie w kodzie pętli).

Uzasadnienie: Poprawa czytelności kodu i łatwości jego utrzymania.

Komentarz 201.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.25 Wyrażenia

Wymaganie Specjalne 202.: Zabrania się używania "operator==" i "operator!=" (bez cudzysłowów) na liczbach rzeczywistych.

Uzasadnienie: Zaokrąglanie i przycinanie w trakcie obliczeń na liczbach rzeczywistych powoduje, że 100% dokładność jest niemożliwa do uzyskania.

Komentarz 202.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 203.: Zabrania się przekręcania liczników (zarówno pow. wart. maks. jak i poniżej wart. min.).

Wyjątek: Algorytmy bazujące na takich zjawiskach. W takich wypadkach szczegółowa dokumentacja jest wymagana.

Uzasadnienie: Przekręcanie liczników zazwyczaj jest sytuacją błędną. Patrz: Zasada 212.

Komentarz 203.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 204.: [Niezrozumiałe - przyp. JMJ] "A single operation with side-effects shall only be used in the following contexts:

1. by itself

2. the right-hand side of an assignment

3. a condition

4. the only argument expression with a side-effect in a function call

5. condition of a loop

6. switch condition

71 W j. ang. for.

7. single part of a chained operation.".

Uzasadnienie: Poprawa czytelności kodu. Patrz: Zasada 204. w Dodatku A.

Komentarz 204.: "side-efect" rozumiem jako sztuczkę jaka robi coś niejawnie. Moim zdaniem nie powinno się wcale takiego kodu programować. Sztuczki warto znać, ale sztuczki w kodzie tylko powodują problemy i doprowadzają do szału zamiast zadziwiać.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 204.1.: Wymaga się by wart. wyrażenia była taka sama niezależenie od kol. parametrów (pod warunkiem, że tak przewiduje standard C++).

Uzasadnienie: BRAK

Wyjątek 204.1.1.: [Gdy w standardzie C++ - przyp. JMJ] podano kolejność wykonania dla danych operatorów i części wyrażeń.

Wyjątek 204.1.2.: [Gdy w standardzie C++ - przyp. JMJ] nie zdefiniowano. Patrz: Zasada 204.1. w Dodatku A.

Komentarz 204.1.: To dobra zasada - ale wyjątki 204.1.1. i 204.1.2. są złe.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 205.: Dopuszcza się dyrektywę ulotna[72] wyłącznie w komunikacji ze sprzętem.

Uzasadnienie: Ta dyrektywa wskazuje kompilatorowi, że wart. zmiennej zmienia się poza programem.

Komentarz 205.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.26 Zarządzanie Pam.

Wymaganie Specjalne 206.: Zabrania się przydzielania pam. na stercie po uruchomieniu [Programu? - przyp. JMJ].

UWAGA: "operator nowy()"[73] może być użyty tylko przy niskopoziomowym zarządzaniu pam. Patrz: Zasada 70.1. kwestie czasu życia obiektu z "operator nowy()".

Uzasadnienie: Intensywne przydzielanie i zwalnianie pam. prowadzi do jej fragmentacji. Dlatego nie da się przewidzieć czasu dostępu do pam. na stercie. Patrz: Alloc.doc[74] w celu poznania alternatywnych rozwiązań.

72 W j. ang. voliate.
73 W j. ang. operator new().
74 Niestety tym dokumentem nie dysponuję - przyp. JMJ.

Komentarz 206.: To dobra zasada - ale nie w cywilu. W cywilu chwila spowolnienia/zacięcia systemu konieczna na defragmentację pam. nie jest problemem. Natomiast widać, że ma to znaczenie w systemie czasu rzeczywistego jakim jest komputer pokładowy myśliwca (F-35).

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 207.: Zabronione są zmienne globalne poza deklaracją kl.

Uzasadnienie: Zmienne globalne są niebezpieczne gdyż brak kontroli dostępu do nich.

Komentarz 207.: To dobra zasada. Jednak by normalnie programować konieczne jest używanie wzorca kl. singleton - w praktyce są to zmienne globalne. Gdy nie używa się wzorca singleton kod jest: nie do wykorzystania w innych projektach, przekomplikowany, nieczytelny i rozwlekły przez długie wywołania potrzebnych f. z wielokrotnie zagnieżdżonych obiektów.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.27 Obsługa Błędów

Wymaganie Specjalne 208.: Zabrania się używania wyjątków.

Uzasadnienie: Za słabe wsparcie narzędzi w obecnych czasach [Czyli w 2005r. bo z tego roku pochodzi dok. "F-35 Coding Rules.pdf" - przyp. JMJ].

Komentarz 208.: To zła zasada. Wtedy może była uzasadniona okolicznościami a nie samą ideą czy implementacją wyjątków w C++.

Nawiasem mówiąc wyjątki są świetne w zgłaszaniu błędów i rozwiązują wiele problemów z tym związanych znanych z j. Asembler i C.

Ocena: Wada.

Zastosowanie: BRAK.

5.28 Przenośność

5.28.1 Postrzeganie Typów Danych

Wymaganie Specjalne 209.: Zabrania się używania typów prostych "int", "short", "long", "float" i "double" (bez cudzysłowów). Zamiast nich należy używać typów z gwarantowaną wielkością w pam. W dodatku dla każdego kompilatora należy je przezwać dyrektywą "typedef" (bez cudzysłowu).

Uzasadnienie: Wielkość typów prostych może się różnić w zależności od kompilatora, a nawet w zależności od platformy sprzętowej. Dlatego należy je przezwać w jednym łatwo dostępnym pliku [Nagłówkowym - przyp. JMJ].

Wyjątek: Niskopoziomowe f. związane z optymalizacją dostępu do danych (np. w zarządzaniu pam.).

Komentarz 209.: To dobra zasada - ale może być lepsza: W trakcie programowania dzieją się rzeczy niesłychane. Np. po kilku latach programowania gdy licznik w projekcie już dawno przebił 100 tyś. linii kodu, okazuje się, że do jakiejś kl. trzeba dodać jedną nową zmienną. Takie zadanie może oznaczać modyfikację setek plików (ROBIŁEM TAKIE RZECZY W PRAKTYCE).

Są proste zasady jakie zabezpieczają przed modyfikowaniem setek plików po drobnej zmianie:

1. Wszystkie kl. używane w programie muszą być przezwane własnym typem;
2. Wszystkie interfejsy z j. C muszą być opakowane własnymi kl.

Wtedy w trakcie całego projektu można swobodnie kontrolować wszystkie używane kl. i typy. Bo gdy przestrzegamy zasady 1. to można łatwo podmienić przedefiniowany typ własną kl. o tej samej nazwie z nowymi zmiennymi jakich na początku projektu nikt nie przewidział. Gdy przestrzegamy zasady 2. zabezpieczamy się przed modyfikacją zarówno używanych bibliotek jak i przed zmianą wymagań w projekcie.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.28.2 Odwzorowanie Typów Danych

Wymaganie Specjalne 210.: Zabrania się programowania algorytmów w sposób bazujący na reprezentacji danych w pam. (np. "big edian" i "litle edian" [Dalsze przykłady są niezrozumiałe - przyp. JMJ] "base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers").

Uzasadnienie: Zapewnienie przenośności kodu.

Wyjątek 210.: Niskopoziomowe operacje (np. przesyłanie danych, konwersja "big edian"<>"litle edian" (bez cudzysłowów) itp.).

Komentarz 210.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 210.1.: Zabrania się programowania algorytmów w sposób bazujący na reprezentacji danych w pam. kl. Patrz: Wymaganie Specjalne 210.

Uzasadnienie: Jest to niezdefiniowane w standardzie C++.

Komentarz 210.1.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 211.: Zabrania się programowania algorytmów w sposób bazujący na adresach zmiennych typu "short", "int", "long", "float", "double" i "long double" (bez cudzysłowów) w obiektach klas.

Uzasadnienie: Reprezentacja danych w pam. jest zależna od sprzętu. Optymalizacje struktur danych mogą powodować przerwy między zmiennymi. Dlatego gdy tej zasady się nie przestrzega kod staje się nieprzenośny.

Wyjątek: Wyjątek 210.

Komentarz 211.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.28.3 Przekręcanie Liczników

Wymaganie Specjalne 212.: Zabrania się bazowania na przekręcaniu liczników w jakikolwiek specjalny sposób[75].

Uzasadnienie: Może być to niezdefiniowane przez standard C++. Bazowanie na niezdefiniowanym zachowaniu jest niedopuszczalne. Patrz: Wymaganie Specjalne 203.

Komentarz 212.: To zła zasada - nic ona nie wnosi. Jest ona zbędna bo Wymaganie Specjalne 203. jednoznacznie zabrania przekręcania liczników.

Ocena: Partactwo.

Zastosowanie: BRAK.

5.28.4 Kolejność Wykonania

Wymaganie Specjalne 213.: Zabrania się polegania na priorytetach operatorów w C++.

Wyjątek: Operatory arytmetyczne.

Uzasadnienie: Poprawa czytelności kodu. Patrz: Zasada 213 w Dodatku A.

75 Tak to chyba trzeba tłumaczyć - przyp. JMJ. W ang. org.: "Underflow or overflow functioning shall not be depended on in any special way.".

Komentarz 213.: To dobra zasada. Operatory arytmetyczne to dobrze wpojona wiedza ze szkoły podstawowej.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Specjalne 214.: Zabrania się robienia założeń dotyczących kolejności tworzenia i przypisania wart. do zmiennych statycznych w innych jednostkach kompilacji[76].

Uzasadnienie: Poleganie na kol. tworzenia i przypisania wart. prowadzi do trudno wykrywalnych błędów.

Komentarz 214.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.28.5 Operacje Na Wskaźnikach

Wymaganie Zwykłe 215.: Zabrania się operacji arytmetycznych na wskaźnikach.

Uzasadnienie: Prowadzi to do wielu błędów. Patrz: Zasada 215. w Dodatku A.

Wyjątek: Kontenery, iteratory i menadżery pamięci, które operują na wsk. za pośrednictwem f.

Komentarz 215.: To dobra zasada.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

5.29 Wydajność Programu

Zalecenie 216.: Zabrania się optymalizacji wczesnych wersji kodu.

Uzasadnienie: Wczesna optymalizacja zaciemnia kod i nie zapewnia eliminacji braku wydajności. Patrz: Zasada 216. w Dodatku A.

"Wczesna optymalizacja to korzeń każdego zła." - Donald Knuth.

UWAGA: Ta zasada nie zabrania doboru elementarnych algorytmów czy struktur danych.

Komentarz 216.: To zła zasada - bo nie o to chodzi. Chodzi o:

Wydajność musi być zapewniona przez architekturę systemu i przez algorytmy, a nie przez sztuczki w kodzie.

76 Tak to chyba trzeba tłumaczyć - przyp. JMJ. W ang. org.: "Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.".

Ocena: Wada.

Zastosowanie: BRAK.

5.30 Różne

Zalecenie 217.: Oczekiwane jest wykrywanie błędów w czasie kompilacji lub linkowania zamiast wykrywania błędów w czasie działania programu.

Gdy wykryje się błąd w czasie wykonania, dodaj jego wykrywanie w momencie kompilacji ew. linkowania - oczywiście wtedy gdy to tylko możliwe.

Uzasadnienie: Błędy wyłapane w trakcie kompilacji i linkowania nie ujawnią się w trakcie działania.

Komentarz 217.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

Wymaganie Zwykłe 218.: Wymaga się konfiguracji ostrzeżeń w opcjach kompilatora zgodnie z założeniami projektu.

Uzasadnienie: Kompilatory mogą zgłaszać pomocne ostrzeżenia o potencjalnych problemach. Może to zaoszczędzić czasu i wysiłku w szukaniu błędów pojawiających się w trakcie wykonania.

Komentarz 218.: To dobra zasada - jednak może być lepsza: należy włączyć wszystkie ostrzeżenia i poprawiać kod tak długo, aż kompilator przestanie narzekać. Jednak problem jest z niezrozumiałymi ostrzeżeniami.

Ocena: Zaleta.

Zastosowanie: Do zapamiętania; Przegląd kodu.

6 Testowanie

Ta sekcja określa jak testować hierarchie dziedziczenia kl. z f. wirt.

6.1.1 Typy Dziedziczące

[W tym miejscu w org. jest powielenie zasady 219 - podanej poniżej - przyp. JMJ]

Wymaganie Specjalne 219.: Wymaga się by kl. potomne przechodziły testy jednostkowe przewidziane dla ich kl. bazowych. Kl. potomne maja silniejsze niezmienniki, więc należy zaprogramować odpowiednie dla nich testy jednostkowe.

Uzasadnienie: Kl. potomne dziedziczące publicznie muszą prawidłowo funkcjonować w kontekstach ich kl. bazowej.

UWAGA 219.: Ta zasada powoduje, że testy kl. bazowych należą do zestawu testów kl. potomnych.

Komentarz 219.: To dobra zasada.

Ocena: Rewelacja.

Zastosowanie: Do zapamiętania; Przegląd kodu.

6.1.2 Struktury

Wymaganie Specjalne 220.: Wymagane jest "spłaszczanie" kl. gdy używa się algorytmu przeglądającego zaw. obiektów.

Uzasadnienie: Przegląd kl. powinno dotyczyć każdej kl. z osobna - nie przypadkiem grup kl. Patrz: Zasada 220. w Dodatku A.

UWAGA: "Spłaszczony widok kl." to wszystkie jej elementy razem ze wszystkim elementami kl. bazowych - bez elementów innych klas.

Komentarz 220.: To dobra zasada. Jednak to wymaganie dla twórców narzędzi programistycznych.

Ocena: Partactwo.

Zastosowanie: BRAK.

Wymaganie Specjalne 221.: Wymaga się przetestowania każdej f. wirt. w każdym możliwym kontekście wywołania: dla kl. do której należy oraz dla wszystkich kl. bazowych.

Uzasadnienie: [Niezrozumiałe - przyp. JMJ] w ang. org.: "Provide decision coverage for dispatch tables.".

Komentarz 221.: To dobra zasada - jednak wykonalna tylko w projektach rządowych finansowanych przez nieograniczone kredyty bankowe.

Ocena: Rewelacja.

Zastosowanie: Testy automatyczne.

7 Bibliografia

Bibliografia

arch-prog-nieup: Jacek Marcin Jaworski, Arch. Prog. Nieuprzywilejowanych, 2025, https://energokod.pl/monografie/Arch.%20Prog.%20Nieuprzywilejowanych.pdf

f-35-arch-s-w-wiki: , Lockheed Martin F-35 Lightning II, 2024, https://pl.wikipedia.org/w/index.php?title=Lockheed_Martin_F-35_Lightning_II&oldid=74520146

umo-na-f-35-podpisana-def24: Janusz Sabak, Umowa na F-35 podpisana, 2020, https://defence24.pl/sily-zbrojne/umowa-na-f-35-podpisana

ntw-2024-11: Michał Gajzler, JASSM niejedno ma imię, czyli AGM-158B-2, B-3, D oraz LRASM i JASSM-XR, 2024

powszechna-deklaracja-praw-człowieka: , Powszechna Deklaracja Praw Człowieka, 1948, https://libr.sejm.gov.pl/tek01/txt/onz/1948.html